AD-A145 093 ADA (TRADEMARK) TRAINING CURRICULUM REAL-TIME CONCEPTS 1/4
L303 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84
DAAB07-83-C-K514

UNCLASSIFIED F/G 9/2 NL

1.0 2.8 2.5 3.2 2.2 3.6 2.0 4.0 1.1 1.8 1.25 1.4 1.6

MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

A145093

①

July 1984

Ada® Training Curriculum

DTIC FILE COPY

AD-A145 093

Real-Time Concepts
L303
Teacher's Guide

84 08 30 104

Center For Tactical Computer Systems
(CENTACS)

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K514

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

L303

REAL-TIME CONCEPTS

L303 -- REAL-TIME CONCEPTS

MODULE OUTLINE

Partial contents:

1. CONCURRENT PROGRAMMING CONCEPTS,

2. Ada TASKING FEATURES,

3. FUNDAMENTAL TASK DESIGNS, and

4. IMPROVING PERFORMANCE.

5. CONCLUSIONS

Accession For	
NTIS GRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

VG 831

B

PREREQUISITES AND GOALS FOR L303

- YOU SHOULD ALREADY HAVE TAKEN MODULE L201 (Ada FOR TECHNICAL MANAGERS) OR HAVE THE EQUIVALENT BACKGROUND

- AFTER COMPLETING L303, YOU SHOULD
  - HAVE A CONCEPTUAL UNDERSTAND OF REAL-TIME AND CONCURRENT Ada PROGRAMMING
  - BE PREPARED TO REVIEW REAL-TIME DESIGNS AND SETTLE DISPUTES
  - *UNDERSTAND WHY Ada IS A VARIABLE LANGUAGE FOR SOLVING REAL-TIME* PROBLEMS

- YOU SHOULD <u>NOT</u> EXPECT TO BE ABLE TO
  - WRITE REAL-TIME PROGRAMS
  - UNDERSTAND SPECIFIC PERFORMANCE-IMPROVEMENT TECHNIQUES

INSTRUCTOR NOTES

ALLOW 90 MINUTES FOR THIS SECTION.

SECTION 1

CONCURRENT PROGRAMMING CONCEPTS

INSTRUCTOR NOTES

THE NOTION OF A PROCESS IS INDEPENDENT OF ANY PROGRAMMING LANGUAGE. IN SECTION 2, AN Ada TASK OBJECT WILL BE DESCRIBED AS CORRESPONDING TO A PROCESS, AN EXCLUSIVE SET OF DATA, AND AN INTERFACE FOR COMMUNICATION WITH OTHER TASKS.

BULLET 3: "CONVENTIONAL PROGRAMS" INCLUDE MOST COMMERCIAL PROGRAMS AND ALL PROGRAMS WRITTEN IN STANDARD FORTRAN.

MULTIPLE PROCESSES

- A PROCESS IS A SEQUENCE OF ACTIONS PERFORMED IN CARRYING OUT A PROGRAM.
- SEVERAL PROCESSES CAN BE IN PROGRESS AT THE SAME TIME.
- "CONVENTIONAL" PROGRAMS ARE WRITTEN FOR A SINGLE PROCESS. THEY SPECIFY ONE SEQUENCE OF ACTIONS TO BE PERFORMED IN A SPECIFIC ORDER.
- IN Ada, YOU CAN WRITE MULTI-PROCESS PROGRAMS. THEY SPECIFY TWO OR MORE SEQUENCES OF ACTIONS THAT MAY BE IN PROGRESS AT THE SAME TIME.
- MODULE L303 CONCENTRATES ON MULTI-PROCESS PROGRAMS.

INSTRUCTOR NOTES

THIS SLIDE MODELS PROGRAMS WITH RECIPES.

THE FIRST "PROGRAM" DESCRIBES EIGHT ACTIONS TO BE PERFORMED IN SEQUENCE TO COOK MACARONI.

THE SECOND "PROGRAM" DESCRIBES TWENTY ACTIONS TO BE PERFORMED TO COOK MACARONI AND CREAM SAUCE. THE MACARONI AND THE CREAM SAUCE CAN BE PREPARED AT THE SAME TIME, SO THIS "PROGRAM" CONSISTS OF TWO PROCESSES. THE ACTIONS LISTED FOR A GIVEN PROCESS MUST BE PERFORMED IN SEQUENCE.

SINGLE- AND MULTIPLE-PROCESS PROGRAMS

- EXAMPLE OF A SINGLE-PROCESS PROGRAM (TO COOK MACARONI):

1. FILL POT WITH WATER.
2. TURN ON HIGH FLAME.
3. WAIT UNTIL WATER IS BOILING.
4. ADD MACARONI TO POT.
5. LOWER FLAME.
6. WAIT UNTIL MACARONI IS COOKED.
7. TURN OFF FLAME.
8. EMPTY CONTENTS OF POT INTO COLANDER.

- EXAMPLE OF A TWO-PROCESS PROGRAM (TO COOK MACARONI AND CREAM SAUCE):

<u>PROCESS 1 (TO COOK MACARONI)</u>:

1. FILL POT WITH WATER.
2. TURN ON HIGH FLAME.
3. WAIT UNTIL WATER IS BOILING.
4. ADD MACARONI TO POT.
5. LOWER FLAME.
6. WAIT UNTIL MACARONI IS COOKED.
7. TURN OFF FLAME.
8. EMPTY CONTENTS OF POT INTO COLANDER.

<u>PROCESS 2 (TO COOK CREAM SAUCE)</u>:

1. PLACE BUTTER IN SAUCEPAN.
2. TURN ON LOW FLAME.
3. WAIT UNTIL BUTTER IS MELTED.
4. ADD FLOUR.
5. MIX WELL.
6. ADD MILK.
7. MIX WELL.
8. RAISE FLAME TO MEDIUM.
9. WAIT UNTIL MIXTURE BOILS, STIRRING OCCASIONALLY.
10. LOWER FLAME
11. STIR CONSTANTLY FOR TWO MINUTES.
12. TURN OFF FLAME.

INSTRUCTOR NOTES

A PROCESSOR IS AN AGENT (SUCH AS A CPU) THAT PERFORMS ACTIONS IN SEQUENCE. IN OVERLAPPED CONCURRENCY, DIFFERENT PROCESSORS PERFORM PROCESSES SIMULTANEOUSLY. IN INTERLEAVED CONCURRENCY, A SINGLE PROCESSOR TAKES TURNS ATTENDING TO SEVERAL PROCESSES AND PERFORMING A FEW ACTIONS FROM EACH IN TURN. EACH PROCESS PROGRESSES WHEN ITS TURN COMES.

IN THE DIAGRAM, EACH STRAIGHT HORIZONTAL LINE CONTAINS THE SEQUENCE OF ACTIONS CONSTITUTING A SINGLE PROCESS. EACH SOLID PATH CONTAINS THE SEQUENCE OF ACTIONS PERFORMED BY A SINGLE PROCESSOR. IN THE ILLUSTRATION OF OVERLAPPED CONCURRENCY, THESE ARE IDENTICAL.

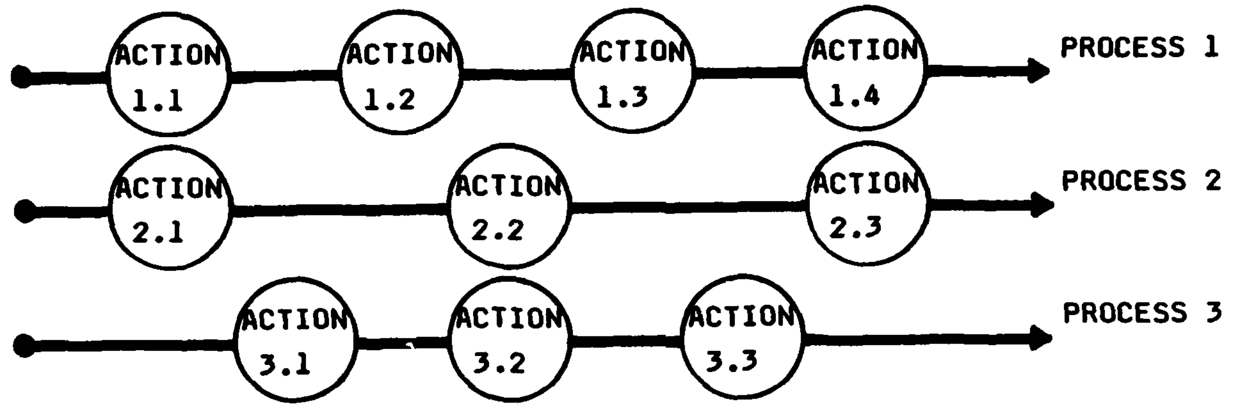
MAKE THE FOLLOWING OBSERVATIONS:

IN INTERLEAVED CONCURRENCY, WHEN ONE PROCESSOR IS BEING SHARED BY SEVERAL PROCESSES, THE PROCESSES PROGRESS MORE SLOWLY.

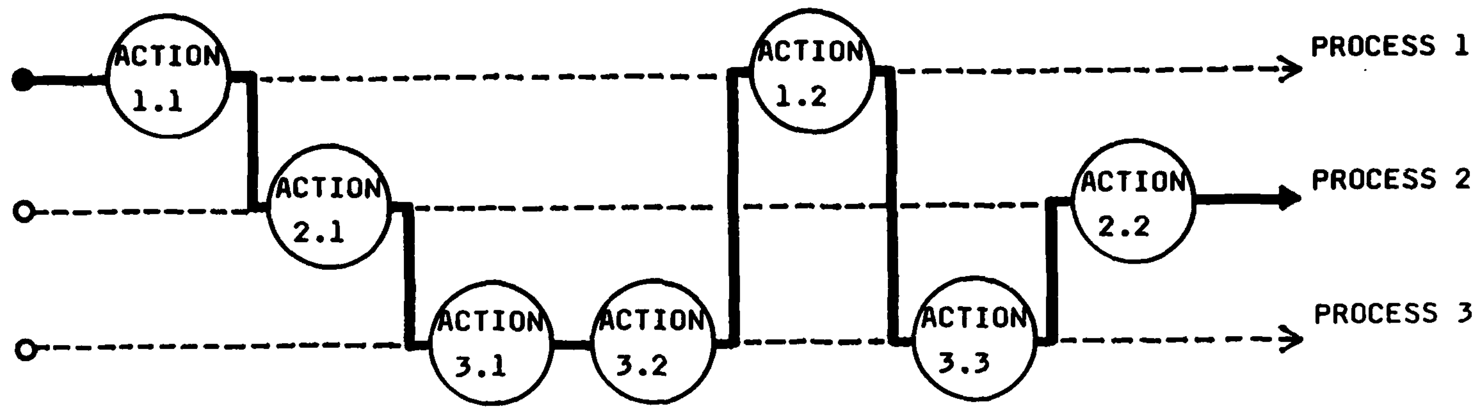
ACTIONS DO NOT OCCUR WITHIN THE SAME ORDER IN BOTH ILLUSTRATIONS (E.G., 1.2 AND 3.2; 2.2 AND 3.3). WITHIN A GIVEN PROCESS, HOWEVER, ACTIONS OCCUR IN RIGID SEQUENCE. (JUST POINT THIS OUT. DETAILED DISCUSSION OF ASYNCHRONISM FOLLOWS LATER.)

OVERLAPPED AND INTERLEAVED CONCURRENCY

- OVERLAPPED CONCURRENCY:



- INTERLEAVED CONCURRENCY:



INSTRUCTOR NOTES

•

THE CHEFS ARE THE PROCESSORS, AND THE PROCESSES ARE THE PREPARATION OF MACARONI AND PREPARATION OF CREAM SAUCE. (SEE THE BOTTOM HALF OF SLIDE 1-2.)

OVERLAPPED AND INTERLEAVED CONCURRENCY -- EXAMPLE

- OVERLAPPED CONCURRENCY:

  TWO CHEFS, ONE COOKING MACARONI WHILE THE OTHER COOKS CREAM SAUCE.

- INTERLEAVED CONCURRENCY:

  ONE CHEF COOKING BOTH MACARONI AND CREAM SAUCE, SWITCHING HIS ATTENTION BACK AND FORTH BETWEEN THE TWO.

- EITHER WAY, THE SET OF DIRECTIONS FOR MACARONI IS CARRIED OUT IN SEQUENCE AND THE SET OF DIRECTIONS FOR CREAM SAUCE IS CARRIED OUT IN SEQUENCE.

INSTRUCTOR NOTES

HERE, ONE CHEF IS COOKING BOTH THE MACARONI AND THE CREAM SAUCE, AS DESCRIBED IN BULLET 2 OF THE PREVIOUS SLIDE.

WHILE ATTENDING TO THE MACARONI, THE CHEF WEARS HIS CHEF JEKYLL HAT. WHEN HE SWITCHES HIS ATTENTION TO THE CREAM SAUCE, HE DONS HIS CHEF HYDE HAT. JUST AS ROBERT LOUIS STEVENSON'S DR. JEKYLL WAS ONE PERSON WHO CREATED THE ILLUSION OF BEING TWO PEOPLE, SO THE CHEF CREATES THE ILLUSION THAT THERE ARE TWO CHEFS -- A CHEF JEKYLL WHO DEVOTES HIS ATTENTION TO MACARONI AND A CHEF HYDE WHO DEVOTES HIS ATTENTION TO CREAM SAUCE.

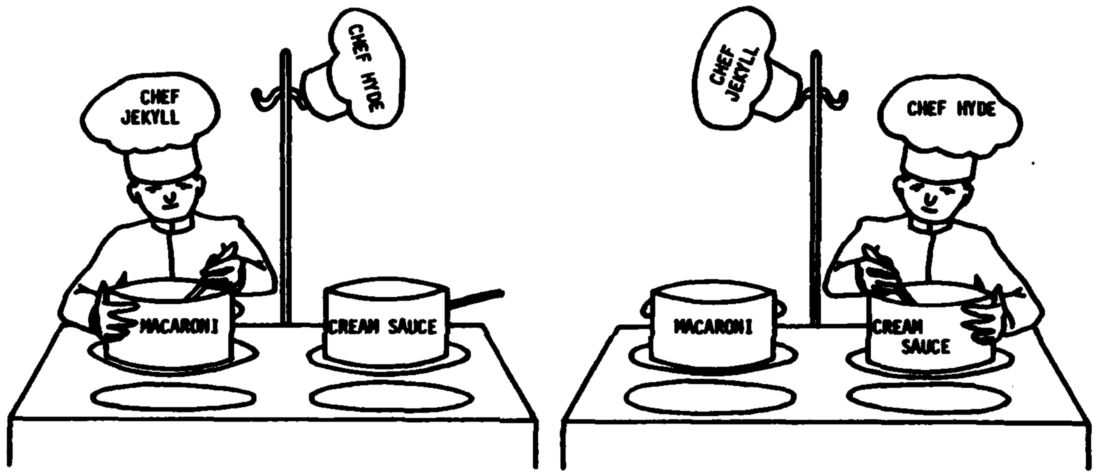
IN COMPUTER SYSTEMS, A PROCESSOR THAT TAKES TURNS EXECUTING DIFFERENT PROCESSES CREATES THE ILLUSION OF SEVERAL SLOWER PROCESSORS EACH DEVOTED TO A SINGLE PROCESS. (IT MAY BE USEFUL TO SHOW SLIDE 1-3 AGAIN TO ILLUSTRATE THIS POINT.) EACH OF THESE ILLUSORY PROCESSORS IS CALLED A <u>VIRTUAL PROCESSOR</u>.

(BULLET 3 SAYS "MAY CORRESPOND" RATHER THAN "CORRESPONDS" BECAUSE OVERLAPPED CONCURRENCY MAY CONSIST OF, SAY, THREE PROCESSORS TAKING TURNS PERFORMING TEN PROCESSES. PHYSICALLY, THREE PROCESSES ARE OVERLAPPED AT ANY MOMENT. LOGICALLY, THERE ARE TEN VIRTUAL PROCESSORS.)

THINKING IN TERMS OF VIRTUAL PROCESSORS ALLOWS A PROGRAMMER TO APPROACH A PROBLEM FROM A HIGHER LEVEL OF ABSTRACTION. THE DISTINCTION BETWEEN OVERLAPPED AND INTERLEAVED CONCURRENCY THEN BECOMES AN IMPLEMENTATION DETAIL.

VIRTUAL PROCESSORS

- INTERLEAVED CONCURRENCY CREATES THE <u>ILLUSION</u> THAT THERE ARE A NUMBER OF SLOWER PROCESSORS, EACH PERFORMING ONE OF THE PROCESSES.



- THE PROCESSORS THAT <u>APPEAR</u> TO EXIST ARE CALLED <u>VIRTUAL PROCESSORS</u>.

- WITH OVERLAPPED CONCURRENCY, EACH VIRTUAL PROCESSOR MAY CORRESPOND TO AN ACTUAL PROCESSOR.

- A PROGRAMMER CAN THINK IN TERMS OF VIRTUAL PROCESSORS WITHOUT WORRYING ABOUT WHETHER CONCURRENCY IS INTERLEAVED OR OVERLAPPED.

INSTRUCTOR NOTES

AN Ada PROGRAM MAY RUN UNDER SOME OPERATING SYSTEM OR ON A BARE MACHINE.

WHEN IT RUNS UNDER AN OPERATING SYSTEM, THE OPERATING SYSTEM PROVIDES MOST OR ALL OF THE SERVICES THAT THE RUNTIME SYSTEM MAKES AVAILABLE TO THE PROGRAM.

WHEN IT RUNS ON A BARE MACHINE, THE RUNTIME SYSTEM ITSELF IMPLEMENTS THESE SERVICES.

BULLET 3 PROVIDES ONLY A PARTIAL LIST OF THE SERVICES PROVIDED BY A RUNTIME SYSTEM. IN L303, WE ARE INTERESTED PRIMARILY IN THE THIRD ITEM LISTED, VIRTUAL PROCESSORS.

THE RUNTIME SYSTEM

- AN Ada PROGRAM RUNS IN AN ENVIRONMENT THAT INCLUDES A <u>RUNTIME SYSTEM</u>.

- A RUNTIME SYSTEM IS THE INTERFACE BETWEEN A RUNNING Ada PROGRAM AND THE UNDERLYING MACHINE OR OPERATING SYSTEM.

- THE RUNTIME SYSTEM PROVIDES SERVICES NEEDED BY THE RUNNING Ada PROGRAM, SUCH AS:

  -- INPUT AND OUTPUT OPERATIONS

  -- ALLOCATION AND DEALLOCATION OF STORAGE

  -- VIRTUAL PROCESSORS TO RUN MULTIPLE PROCESSES

- THE RUNTIME SYSTEM IS "THE EXECUTIVE."

INSTRUCTOR NOTES

BULLET 1:

ACHIEVING INTERLEAVING:

AN IN-DEPTH DISCUSSION OF PROCESSOR SCHEDULING IS BEYOND THE SCOPE OF THIS MODULE.

BULLETS 2-4:

THE RULES OF Ada REQUIRE THE RUNTIME SYSTEM TO PROVIDE VIRTUAL PROCESSORS, BUT DO NOT CONSTRAIN HOW THIS IS DONE.

THE RULES OF Ada CAN BE UNDERSTOOD IN TERMS OF VIRTUAL PROCESSORS WITHOUT UNDERSTANDING THE UNDERLYING IMPLEMENTATION.

ALTERNATIVE RUNTIME SYSTEMS

- THERE ARE MANY DIFFERENT WAYS TO BUILD RUNTIME SYSTEMS.
  - -- ALTERNATIVE WAYS TO PROVIDE VIRTUAL PROCESSORS
    - OVERLAPPED CONCURRENCY WITH EACH VIRTUAL PROCESSOR CORRESPONDING TO A PHYSICAL PROCESSOR.
    - INTERLEAVED CONCURRENCY ON A SINGLE PHYSICAL PROCESSOR.
    - A COMBINATION, E.G., INTERLEAVING FIVE PROCESSES ON TWO OVERLAPPED PROCESSORS.
  - -- ALTERNATIVE RULES FOR DETERMINING WHICH PROCESS A PROCESSOR ATTENDS TO AT A GIVEN TIME
  - -- ETC., ETC., ETC.
- THE RULES OF Ada CAN BE MET BY ANY OF THESE CHOICES.
- THE INNER WORKINGS OF THE RUNTIME SYSTEM ARE HIDDEN FROM THE PROGRAMMER.
- AN Ada PROGRAM CAN USUALLY BE WRITTEN SO THAT ITS LOGIC DOES NOT DEPEND ON THE INNER WORKINGS OF THE RUNTIME SYSTEM.

INSTRUCTOR NOTES

BULLET 1:

A CORRECT PROGRAM MAY PRODUCE DIFFERENT RESULTS WITH DIFFERENT RUNTIME SYSTEMS. BY "VALID" WE MEAN THAT ANY OF THESE RESULTS ARE ACCEPTABLE.

(THE FIRST BULLET ASSUMES THAT ALL THE RUNTIME SYSTEMS FOR A GIVEN COMPILER HAVE THE SAME INTERFACE IN TERMS OF Ada. FOR EXAMPLE, THE MEANING OF LOW-LEVEL FEATURES IS ASSUMED TO BE THE SAME, NO MATTER WHICH RUNTIME SYSTEM THE COMPILER IS WORKING WITH. THUS CHANGING A RUNTIME SYSTEM SHOULD NOT NECESSITATE CHANGING Ada SOURCE TEXT, EXCEPT PERHAPS TO IMPROVE PERFORMANCE.)

BULLET 4:

CUSTOMIZING OR REWRITING AN Ada RUNTIME SYSTEM IS EQUIVALENT TO WRITING AN EXECUTIVE AS THE FIRST STEP IN IMPLEMENTING A REAL-TIME SYSTEM.

DIFFERENT RUNTIME SYSTEMS ARE BETTER FOR DIFFERENT APPLICATIONS

- Ada PROGRAMS CAN BE WRITTEN TO PRODUCE EQUALLY VALID <u>ANSWERS</u> WITH ANY RUNTIME SYSTEM.
- THE CHOICE OF A RUNTIME SYSTEM MAY PROFOUNDLY AFFECT THE <u>PERFORMANCE</u> OF THE PROGRAM.
- DIFFERENT RUNTIME SYSTEMS, WITH DIFFERENT PERFORMANCE CHARACTERISTICS, MAY BE AVAILABLE FOR THE SAME TARGET MACHINE.
- FOR SOME PROJECTS IT MAY BE NECESSARY TO CUSTOM-BUILD A RUNTIME SYSTEM, OR TO MODIFY AN EXISTING ONE.
- THE GOOD NEWS:
  - THIS MAKES IT EASIER TO TAILOR SYSTEMS WRITTEN IN Ada TO PROJECT REQUIREMENTS.
  - AS TIME GOES ON, MORE AND MORE RUNTIME SYSTEMS WILL BE AVAILABLE OFF THE SHELF.

INSTRUCTOR NOTES

- BULLET 1: THE NEXT SLIDE EXPLAINS WHY THIS IS SO.

- BULLET 3:

  IF ONE PROCESS PRODUCES DATA THAT IS USED BY THE SECOND PROCESS, SPECIAL MEASURES ARE NECESSARY TO ENSURE THAT THE ACTIONS OF EACH PROCESS OCCUR IN THE NECESSARY ORDER. IF TWO PROCESSES UPDATE THE SAME DATA, SPECIAL MEASURES ARE NECESSARY TO ENSURE THAT THE PROCESSES DO NOT INTERFERE WITH EACH OTHER.

  WE EXAMINE PROBLEMS INTRODUCED BY THIS ASYNCHRONISM MORE CLOSELY TOWARDS THE END OF THIS SECTION.

ASYNCHRONOUS PROCESSES

- PROCESSES ARE GENERALLY ASYNCHRONOUS. THAT IS, THEY PROCEED AT DIFFERENT RATES.
- THE RELATIVE PROGRESS OF ONE PROCESS WITH RESPECT TO ANOTHER IS UNPREDICTABLE.
- THIS CAN CREATE PROBLEMS WHEN ONE PROCESS TRIES TO USE DATA BEING PRODUCED BY ANOTHER.
- THERE ARE WAYS TO CAUSE PROCESSES TO SYNCHRONIZE MOMENTARILY.

INSTRUCTOR NOTES

- BULLET 2:

  -- ITEM 2: FOR EXAMPLE, PROCESSES MAY HAVE DIFFERENT PRIORITIES

- BULLET 3:

EXTERNAL INPUTS CAN MAKE IT IMPOSSIBLE IN PRINCIPLE TO PREDICT THE RELATIVE SPEEDS OF DIFFERENT PROCESSES. EVEN IF THIS WERE NOT THE CASE, THE FACTORS INVOLVED ARE SO COMPLEX THAT PREDICTION WOULD BE IMPOSSIBLE IN PRACTICE.

BY REGARDING THE RELATIVE PROGRESS OF DIFFERENT PROCESSES AS ESSENTIALLY RANDOM, A PROGRAMMER GREATLY REDUCES THE NUMBER OF DETAILS WITH WHICH HE HAS TO BE CONCERNED.

THIS "RANDOM FACTOR" IS ABSENT FROM MOST SEQUENTIAL PROGRAMS. ONE OF THE BIGGEST HURDLES A SEQUENTIAL PROGRAMMER FACES WHEN LEARNING CONCURRENT PROGRAMMING IS RECOGNIZING AND COPING WITH NON-DETERMINISM.

WHY PROCESSES ARE ASYNCHRONOUS

- THEY MAY BE RUNNING ON PHYSICAL PROCESSORS WITH DIFFERENT SPEEDS.
- THEY MAY BE RUNNING ON VIRTUAL PROCESSORS IMPLEMENTED BY INTERLEAVING.
  - THE AMOUNT OF TIME SPENT ON ONE PROCESS BEFORE SWITCHING TO ANOTHER MAY NOT BE UNIFORM.
  - SELECTION OF THE NEXT PROCESS TO BE PERFORMED MAY NOT TREAT ALL PROCESSES EQUALLY.
  - PROCESSES MAY VOLUNTARILY SUSPEND THEMSELVES UNTIL THE OCCURRENCE OF SOME EXTERNAL EVENT:
    - AN INTERRUPT
    - PASSAGE OF A SPECIFIED AMOUNT OF TIME
    - THE PERFORMANCE OF SOME ACTION BY ANOTHER PROCESS
- TIMING DEPENDS NOT ONLY ON THE RUNTIME SYSTEM, BUT ALSO ON THE EXTERNAL INPUTS DURING A GIVEN EXECUTION.

INSTRUCTOR NOTES

- BULLET 2:

  ACTION 2.2 MAY OCCUR BEFORE 1.2, BETWEEN 1.2 AND 1.3, OR AFTER 1.3, BUT IT MUST OCCUR AFTER 1.1 AND BEFORE 1.4.

- BULLET 3:

  BY FORCING A PROCESS THAT GETS AHEAD TO WAIT FOR A PROCESS THAT FALLS BEHIND, SYNCHRONIZATION CAN CAUSE PROCESSES TO MANIFEST THE SAME <u>AVERAGE</u> SPEED <u>IN THE LONG RUN</u>.

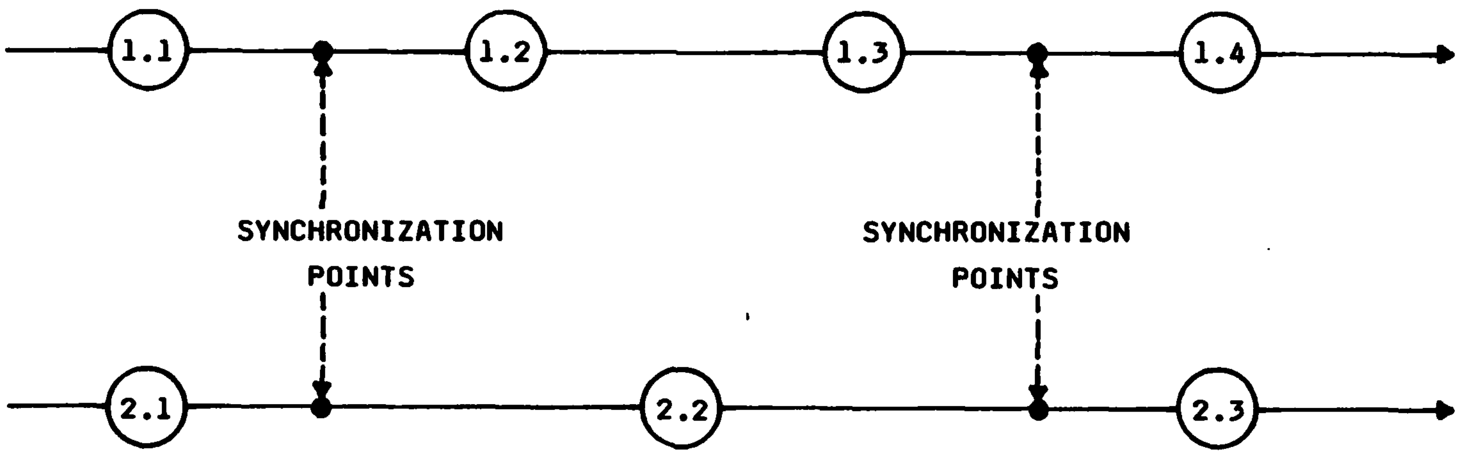
- BULLET 4:

  Ada'S MECHANISMS FOR SYNCHRONIZATION ARE EXPLAINED IN SECTION 2.

SYNCHRONIZATION

- SYNCHRONIZATION IS A WAY TO ASSURE THAT ONE PROCESS IS AT A SPECIFIED POINT AT THE SAME TIME THAT THE OTHER PROCESS IS AT A SPECIFIED POINT. (THESE POINTS ARE CALLED SYNCHRONIZATION POINTS.)

- SYNCHRONIZATION REDUCES THE NUMBER OF WAYS TWO PROCESSES CAN BE INTERLEAVED.



- SYNCHRONIZATION DOES NOT CONTROL THE RELATIVE SPEEDS OF PROCESSES <u>BETWEEN</u> SYNCHRONIZATION POINTS, BUT FORCES A FAST PROCESS TO WAIT AT A SYNCHRONIZATION POINT WHILE A SLOW ONE CATCHES UP.

- PROCESSES MUST SYNCHRONIZE TO INTERACT IN A PREDICTABLE WAY.

INSTRUCTOR NOTES

"SYNCHRONIZING WITH THE CLOCK" MEANS GUARANTEEING THAT A PROCESS WILL BE AT A CERTAIN POINT WHEN THE CLOCK IS AT A CERTAIN POINT. IT IS DIFFERENT FROM SYNCHRONIZING WITH AN ORDINARY PROCESS, BECAUSE THE CLOCK CAN'T BE MADE TO WAIT WHILE ANOTHER PROCESS CATCHES UP.

CONCURRENT PROGRAMMING AND REAL-TIME PROGRAMMING

- CONCURRENT PROGRAMMING IS THE CONSTRUCTION OF A PROGRAM SPECIFYING ACTIONS FOR MULTIPLE PROCESSES.
    - IN CONCURRENT PROGRAMMING, COOPERATING PROCESSES MUST SYNCHRONIZE WITH EACH OTHER.
- REAL-TIME PROGRAMMING IS A SPECIAL FORM OF CONCURRENT PROGRAMMING IN WHICH ACTIONS MUST BE PERFORMED WITHIN SPECIFIED TIME INTERVALS.
    - PROCESSES MUST SYNCHRONIZE NOT ONLY WITH EACH OTHER, BUT WITH THE CLOCK.

INSTRUCTOR NOTES

THE THREE PEOPLE FOLLOWING THE SAME INSTRUCTIONS IN THIS PICTURE REPRESENT MULTIPLE PROCESSES EXECUTING THE SAME PROGRAM.

JUST AS ONE PERSON MAY STILL BE ON INSTRUCTION 1 WHILE ANOTHER HAS MOVED ON TO INSTRUCTION 2, SO DIFFERENT PROCESSES CAN EXECUTE THE SAME PROGRAM AT DIFFERENT RATES.

- BULLET 4:

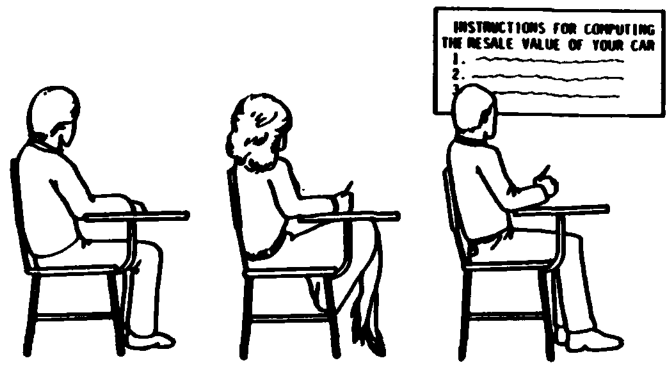
  EACH PERSON IN THE DRAWING IS PERFORMING THE NECESSARY CALCULATIONS ON HIS OWN SCRATCHPAD. THAT IS WHAT ENABLES THEM TO FOLLOW THE SAME INSTRUCTIONS AT THE SAME TIME WITHOUT PAYING ANY ATTENTION TO EACH OTHER.

  IF THE INSTRUCTIONS CALLED FOR WRITING FIGURES ON THE SHARED BLACKBOARD AND LATER READING THOSE FIGURES -- OR WORSE YET, IF ONE INSTRUCTION CALLED FOR CHANGING ANOTHER INSTRUCTION -- THE THREE PEOPLE WOULD INTERFERE WITH EACH OTHERS' CALCULATIONS.

  A PROGRAM THAT DOES NOT MODIFY ITSELF AND THAT KEEPS DATA TO BE MANIPULATED SEPARATE FROM INSTRUCTIONS IS CALLED <u>REENTRANT</u>. Ada COMPILERS ONLY PRODUCE REENTRANT INSTRUCTIONS, SO ANY Ada SUBPROGRAM (FOR EXAMPLE) CAN BE CALLED BY ONE PROCESS WHILE IT IS STILL BEING EXECUTED BY ANOTHER PROCESS.

PROGRAMS VERSUS PROCESSES

- A PROGRAM IS A SET OF <u>INSTRUCTIONS</u>.
- A PROCESS IS A SET OF <u>ACTIONS</u> CARRIED OUT IN ACCORDANCE WITH INSTRUCTIONS.
- TWO PROCESSES CAN FOLLOW DIFFERENT SETS OF INSTRUCTIONS, OR THEY CAN FOLLOW THE SAME SET OF INSTRUCTIONS, EACH AT ITS OWN PACE.
- IN Ada, THE RUNTIME SYSTEM SUPPLIES A SEPARATE <u>DATA AREA</u> WITH EACH VIRTUAL PROCESSOR, SO EACH PROCESS KEEPS ITS OWN COPY OF THE VARIABLES USED BY THE INSTRUCTIONS.



INSTRUCTOR NOTES

THE NEXT FOUR SLIDES EXPAND UPON THESE FOUR REASONS FOR CONCURRENCY.

SOME OF THE REASONS ARE INTERRELATED.

SOME REASONS FOR CONCURRENCY

- MANAGEMENT OF SIMULTANEOUS REAL-WORLD ACTIVITIES.
- SIMULATION OF SIMULTANEOUS REAL-WORLD ACTIVITIES.
- PARALLEL COMPUTATION TO INCREASE THROUGHPUT.
- LOGICAL DECOMPOSITION OF A COMPLEX PROBLEM INTO SIMPLE PROCESSES.

INSTRUCTOR NOTES

BULLET 1 DESCRIBES ACTIVITIES GOING ON IN THE REAL WORLD, BULLET 2 DESCRIBES THE CHORES THE PROGRAM MUST PERFORM IN RELATION TO THESE ACTIVITIES. THE CORRESPONDENCE IS AS FOLLOWS:

ACTIVITY 2 ---------------------- CHORE 1

ACTIVITY 3 ---------------------- CHORE 2

ACTIVITIES 4 AND 1 --------------- CHORE 3 (ONE INSTANCE FOR EACH ZONE)

ACTIVITY 5 ---------------------- CHORE 4

ACTIVITIES 4 AND 1 ARE COMBINED BECAUSE MONITORING THE TEMPERATURE AND CONTROLLING THE VENT IN A GIVEN ZONE ARE CLOSELY SYNCHRONIZED ACTIVITIES. EXCEPT FOR THIS, THE REAL-WORLD ACTIVITIES ARE NOT SYNCHRONIZED WITH EACH OTHER. THAT IS ONE REASON THAT IT IS APPROPRIATE TO ASSIGN THE VARIOUS CHORES TO ASYNCHRONOUS PROCESSES.

MANAGEMENT OF SIMULTANEOUS REAL-WORLD ACTIVITIES

- SIMULTANEOUS REAL-WORLD ACTIVITIES FOR A MULTI-ZONE HEATING SYSTEM:
  - TEMPERATURE FLUCTUATES IN EACH ZONE.
  - TIME OF DAY CHANGES.
  - AT ARBITRARY TIMES, AN OPERATOR KEYS IN DESIRED TEMPERATURE FOR A GIVEN ZONE FOR A GIVEN TIME OF DAY.
  - VENTS FOR EACH ZONE ARE OPENED AND CLOSED.
  - HEATER IS SET TO OFF, LOW, MEDIUM, OR HIGH, DEPENDING ON NUMBER OF OPEN VENTS.

- CONCEPTUALLY, THE PROGRAM MUST SIMULTANEOUSLY PERFORM THE FOLLOWING CHORES:
  - KEEP TRACK OF THE TIME OF DAY.
  - INTERPRET AND ACT UPON KEYPAD INPUT.
  - CONTROL EACH ZONE'S VENT BASED ON CURRENT ZONE TEMPERATURE, TIME OF DAY, AND CURRENT DESIRED TEMPERATURE FOR THE ZONE.
  - CONTROL THE HEATER SETTING.

- A SEPARATE PROCESS CAN BE CREATED FOR EACH OF THESE CHORES.
  - EACH CHORE HAS ITS OWN SEQUENCE OF INSTRUCTIONS.
  - THE PROCESSES MIRROR THE REAL-WORLD ACTIVITIES.

INSTRUCTOR NOTES

- BULLET 1: ITEMS 3 AND 4 ARE REAL-TIME SIMULATIONS. ITEMS 1 AND 2 ARE NOT.

  - ITEM 1: THE SIMULATION MAY INCLUDE RANDOM NUMBER GENERATION WITH A SPECIFIED PROBABILITY DISTRIBUTION.

  - ITEM 2: DIFFERENT TIMING AND SYNCHRONIZATION OF TRAFFIC LIGHTS CAN BE SIMULATED IN SEARCH FOR A SYSTEM THAT WILL MANIFEST THE DESIRED BEHAVIOR. IT IS LESS EXPENSIVE TO MODIFY THE PROTOTYPE BASED ON SIMULATION THAN TO MODIFY A FULLY IMPLEMENTED SYSTEM BASED ON EXPERIENCE.

  - ITEM 3: IN THE SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY (SREM), A SIMULATION IS RUN BASED ON SOFTWARE <u>REQUIREMENTS</u>, TO TEST WHETHER PERFORMANCE REQUIREMENTS ARE <u>FEASIBLE</u>. IN THE CASE OF THE RADAR SYSTEM, THE INPUTS TO THE PROGRAM ARE BEING SIMULATED. IN THE CASE OF SREM, EXECUTION OF THE PROPOSED PROGRAM ITSELF IS SIMULATED.

  - ITEM 4: SIMULATORS CAN BE USED TO TRAIN PEOPLE TO USE NEW EQUIPMENT. THE SIMULATOR SIMULATES THE BEHAVIOR OF THE EQUIPMENT.

- BULLET 2: THE RULES FOR A GIVEN ENTITY MAY DEPEND ON THE CURRENT STATE OF OTHER ENTITIES.

- BULLET 3: THERE IS A ONE-TO-ONE CORRESPONDENCE BETWEEN ENTITIES BEING MODELED AND PROCESSES.

SIMULATING SIMULTANEOUS REAL-WORLD ACTIVITIES

- COMMON USES OF SIMULATION:
  - PREDICTION OF PHYSICAL PHENOMENA
    - PREDICTION BASED ON AN ASSUMED MODEL (SET OF RULES)
    - MODEL MAY BE TOO COMPLEX FOR MATHEMATICAL ANALYSIS
    - EXAMPLES: NUCLEAR REACTOR BEHAVIOR, AERODYNAMICS
  - PROTOTYPING
    - BUILD A PROGRAM SIMULATING A PROPOSED HARDWARE/SOFTWARE SYSTEM
    - OBSERVE THE SYSTEM'S BEHAVIOR AND REFINE THE RULES BEFORE IMPLEMENTING THE SYSTEM
    - EXAMPLE: CENTRAL TRAFFIC SIGNAL CONTROL SYSTEM
  - SOFTWARE TESTING
    - TEST EMBEDDED COMPUTER SOFTWARE IN THE LAB BEFORE INSTALLING IT IN A TANK, AIRCRAFT, OR MISSILE.
    - EXTERNAL INPUTS ARE SIMULATED.
    - EXAMPLE: IN A RADAR SYSTEM, SIMULATE AIRCRAFT MOVING IN TRACKS AND GENERATING ECHOES.
  - TRAINING
    - BUILD A PROGRAM SIMULATING THE SYSTEM BEHIND THE USER INTERFACE.
    - BOTH EXTERNAL EVENTS AND RESPONSES TO USER INPUTS CAN BE SIMULATED.
    - EXAMPLE: COCKPIT SIMULATOR
- IN EACH CASE, SIMULATION IS BASED ON RULES FOR BEHAVIOR OF VARIOUS INDIVIDUAL ENTITIES IN THE MODEL.
- INTERACTIONS ARE COMPLEX, BUT EACH ENTITY CAN BE MODELED BY A PROCESS DIRECTLY IMPLEMENTING THE RULES FOR THAT ENTITY.

INSTRUCTOR NOTES

BULLET 3: THE TOTAL AMOUNT OF COMPUTATION TIME ON ALL PROCESSORS MAY BE THE SAME, BUT MORE THAN ONE PROCESSOR CAN BE EXPENDING THIS TIME. (THE SAME NUMBER OF PROCESSOR-SECONDS IN FEWER SECONDS.)

BULLET 4: OTHERWISE, THE PROCESSOR WOULD REMAIN IDLE WHILE WAITING FOR THE EXTERNAL EVENT TO OCCUR.

PARALLEL COMPUTATION TO INCREASE THROUGHPUT

- SOMETIMES PARTS OF A COMPUTATION CAN BE DECOMPOSED INTO STEPS THAT DO NOT DEPEND ON EACH OTHERS' RESULTS.

- THESE STEPS CAN BE EXECUTED CONCURRENTLY BY DIFFERENT PROCESSES.

- IF EACH PROCESS IS BEING EXECUTED ON A DIFFERENT PHYSICAL PROCESSOR, THE COMPUTATION CAN COMPLETE MORE QUICKLY.

- IF CERTAIN PROCESSES MUST OCCASIONALLY WAIT FOR EXTERNAL EVENTS TO OCCUR, A SINGLE PROCESSOR CAN COMPLETE THE JOB MORE QUICKLY BY WORKING ON ONE PROCESS WHILE THE OTHER PROCESS IS WAITING.
  - EXAMPLE: SOME PROBLEMS CAN BE SOLVED USING AN INPUT PROCESS, A COMPUTATION PROCESS, AND AN OUTPUT PROCESS. THE COMPUTATION PROCESS CAN PROCEED WHILE THE OTHER TWO PROCESSES ARE WAITING FOR COMPLETION OF AN *I/O* OPERATION.

INSTRUCTOR NOTES

THIS USE OF CONCURRENT PROCESSES IS REVISITED IN SECTION 3.

- BULLET 1:

  THIS REFORMATTING ALLOWS TEXT ORIGINALLY FORMATTED FOR 1-INCH MARGINS AND PICA TYPE TO BE PRINTED WITH 1-INCH MARGINS AND ELITE TYPE (ASSUMING STANDARD-WIDTH PAPER).

- BULLET 2:

  - TRANSFORMATION 1: THE EXTRA SPACES ARE TO SEPARATE THE LAST WORD OF ONE LINE FROM THE FIRST WORD OF THE NEXT LINE IN THE STREAM OF CHARACTERS.

  - TRANSFORMATION 2: FOR THE PURPOSES OF THIS EXAMPLE, ANY GROUP OF CONSECUTIVE NON-BLANKS FORMS A WORD.

- BULLET 3:

  AT FIRST, IT MAY BE EASIER TO UNDERSTAND THE TRANSFORMATIONS AS BEING RUN ONE AT A TIME, WITH THE OUTPUT STREAM OF ONE GOING INTO AN INTERMEDIATE FILE THAT IS THEN USED FOR THE INPUT STREAM OF THE NEXT TRANSFORMATION.

  IN FACT, THERE IS NO NEED TO WAIT FOR ONE TRANSFORMATION TO FINISH BEFORE STARTING THE NEXT ONE. WE CAN THINK OF THE TRANSFORMATIONS AS OCCURRING CONCURRENTLY, CONSUMING DATA FROM AN INPUT STREAM AND PRODUCING DATA IN AN OUTPUT STREAM. OF COURSE, THE PROCESSES PERFORMING THESE TRANSFORMATIONS MAY SPEND A LOT OF TIME WAITING FOR DATA TO MOVE ALONG THE STREAM.

- BULLET 4:

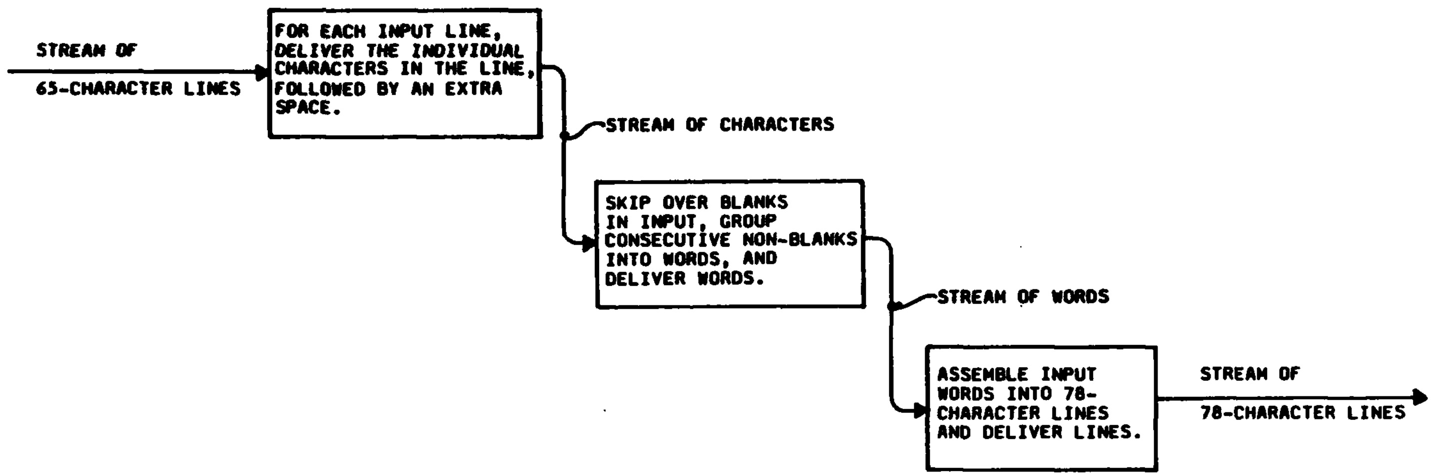
  THIS IS THE BASIC THRUST OF THE JACKSON STRUCTURED DESIGN METHODOLOGY. WHEN THE INPUT STREAM AND OUTPUT STREAM HAVE INCOMPATIBLE STRUCTURES, TRANSFORMATIONS ARE HARD TO PROGRAM BECAUSE THEY MUST MANAGE TWO CONCEPTUAL THREADS AT ONCE, ONE DEALING WITH THE STRUCTURE OF EACH STREAM.

LOGICAL DECOMPOSITION OF A COMPLEX PROBLEM

- PROBLEM:

  REFORMAT 65-COLUMN LINES OF TEXT INTO 78-COLUMN LINES, FITTING AS MANY WORDS ON A LINE A POSSIBLE.

- A SOLUTION IN TERMS OF THREE SIMPLE TRANSFORMATIONS:



- THE TRANSFORMATIONS NEED NOT RUN ONE AFTER THE OTHER. THEY CAN BE WRITTEN AS CONCURRENT PROCESSES.

- EACH PROCESS IS EASY TO PROGRAM BECAUSE IT IMPLEMENTS A SIMPLE TRANSFORMATION. EACH TRANSFORMATION IS SIMPLE BECAUSE THE STRUCTURE OF ITS INPUT IS CLOSELY RELATED TO THE STRUCTURE OF ITS OUTPUT.

INSTRUCTOR NOTES

THIS SLIDE SUMMARIZES THE COMMON THEMES OBSERVED FOR ALL FOUR USES OF CONCURRENCY.

BULLET 4: CONCURRENCY IS A USEFUL <u>CONCEPTUAL MODEL</u> FOR SOLVING A PROBLEM WITH MANY THREADS OF ACTIVITY.

IT IS NOT A LOW-LEVEL IMPLEMENTATION DETAIL.

COMMON THEMES

- SEVERAL PROCESSES MIRROR THE EXISTENCE OF SEVERAL CONCEPTUAL THREADS:

  - REAL-WORLD ACTIVITIES
  - ENTITIES BEING SIMULATED
  - LOGICALLY INDEPENDENT PROCESSING STEPS
  - TRANSFORMATIONS ACTING ON DATA STREAMS

- DETAILS OF SCHEDULING AND INTERLEAVING ARE HANDLED AUTOMATICALLY BY THE RUNTIME SYSTEM, AND DO NOT APPEAR IN THE PROGRAM.

- THE STRUCTURE OF THE PROGRAM REFLECTS THE CONCEPTUAL THREADS.

- MULTI-PROCESS PROGRAMMING IS MORE THAN A WAY OF SPECIFYING THAT CERTAIN ACTIONS CAN BE EXECUTED SIMULTANEOUSLY. IT IS A WAY OF THINKING ABOUT THE STRUCTURE OF A PROBLEM.

INSTRUCTOR NOTES

THE NEXT FEW SLIDES DESCRIBE THE PROBLEMS LISTED IN BULLET 2, ILLUSTRATING EACH WITH A CARTOON.

PROCEED WITH CAUTION!

- CONCURRENT PROGRAMMING IS TRICKY.
- IT ENTAILS MANY SUBTLE PROBLEMS THAT DO NOT ARISE IN SINGLE-PROCESS PROGRAMMING, INCLUDING:
  - SIMULTANEOUS UPDATE OF DATA BY MORE THAN ONE PROCESS
  - DEADLOCK
  - STARVATION
  - SYNCHRONIZATION AND COMMUNICATION AMONG PROCESSES
- INTUITION DEVELOPED THROUGH YEARS OF SINGLE-PROCESS PROGRAMMING IS NOT SUFFICIENT.

INSTRUCTOR NOTES

- BULLET 2:

  FOR EXAMPLE, IF PROCESS 1 AND PROCESS 2 BOTH EXAMINE AN OBJECT, THEN PROCESS 1 UPDATES IT, PROCESS 2 MAY THEN UPDATE THE OBJECT BASED ON ITS PREVIOUS STATE. IN A RACE CONDITION, THE OUTCOME OF A COMPUTATION MAY DEPEND ON PROCESS TIMING.

- BULLET 3:

  A PROCESS TRYING TO GAIN EXCLUSIVE ACCESS WHILE ANOTHER PROCESS ALREADY HAS IT IS FORCED TO WAIT UNTIL THE OTHER PROCESS RELINQUISHES IT.

- THE CARTOON:

  SOUTHERN PATHETIC RAILROAD HAS TWO PARALLEL TRACKS, ONE FOR TRAINS GOING IN EACH DIRECTION.

  HOWEVER, THERE IS ONLY ONE TRACK CROSSING THE RIVER, SHARED BY TRAINS GOING IN BOTH DIRECTIONS.

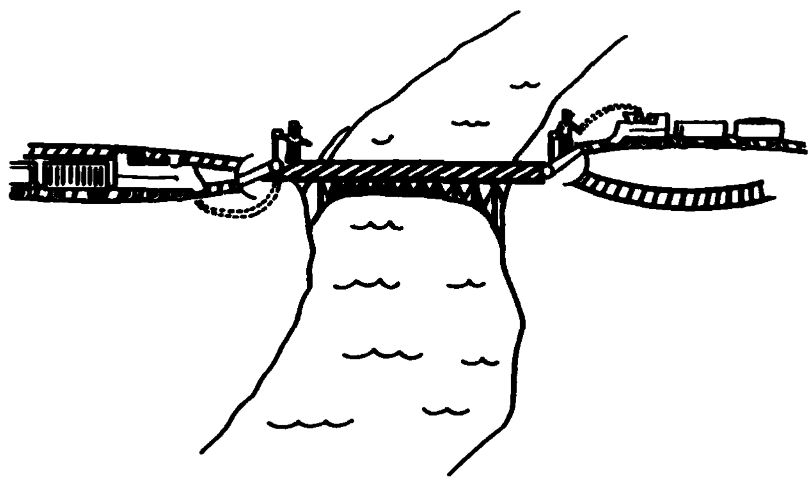
  SWITCHES AT THE ENDS OF THE SHARED TRACK CONNECT IT WITH EITHER OF THE TWO PARALLEL TRACKS.

  IN THIS SCENE, EACH ENGINEER HAS COME TO THE SHARED RAIL, OBSERVED THAT IT WAS NOT (YET) IN USE BY ONCOMING TRAINS, AND SWITCHED THE SHARED TRACK TO HIS DIRECTION. EACH ENGINEER WILL NOW CLIMB BACK INTO THE TRAIN AND RESUME HIS TRIP.

  WHEN THEY MEET IN THE MIDDLE OF THE BRIDGE, THE ENGINEERS WILL LEARN ABOUT THE HAZARDS OF SIMULTANEOUS UPDATE THE HARD WAY.

SIMULTANEOUS UPDATE

- TWO OR MORE PROCESSES EXAMINING AND THEN MODIFYING THE SAME OBJECT.
- RESULTS OF EXAMINATION ARE UNRELIABLE BECAUSE ANOTHER PROCESS MIGHT MODIFY THE OBJECT IMMEDIATELY AFTERWARD. THIS IS CALLED A RACE CONDITION.
- THE SOLUTION IS MUTUAL EXCLUSION:
  - A PROCESS GAINS EXCLUSIVE USE OF AN OBJECT BEFORE EXAMINING IT AND RELINQUISHES EXCLUSIVE ACCESS AFTER MODIFYING IT.



TO AVOID A REPETITION OF THE DISASTER DESCRIBED ON THE PREVIOUS SLIDE, SOUTHERN PATHETIC ISSUED THE FOLLOWING DIRECTIVE TO ITS ENGINEERS:

"UPON COMING TO THE SHARED TRACK, TAKE OUT YOUR TELESCOPE AND CHECK WHETHER THERE IS A TRAIN APPROACHING IN THE OPPOSITE DIRECTION. IF THERE IS, DO NOT PROCEED UNTIL THAT TRAIN HAS PASSED."

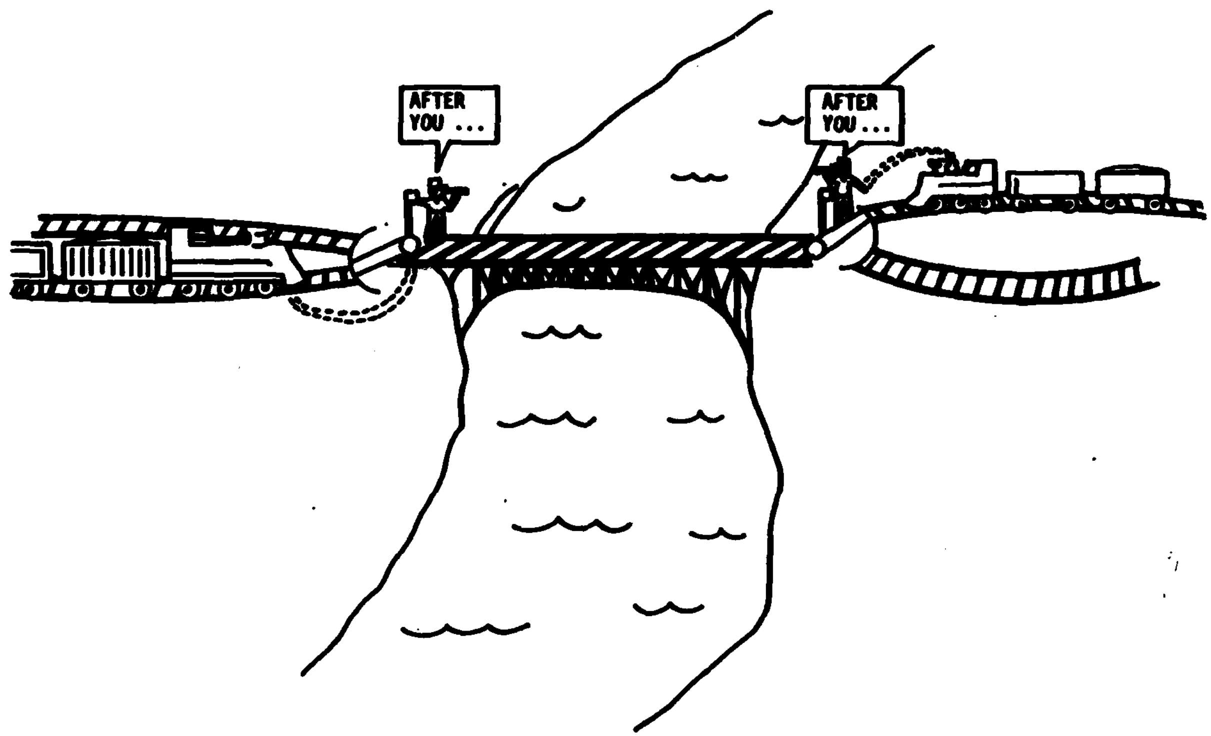
THIS SLIDE DEPICTS BOTH ENGINEERS FASTIDIOUSLY ADHERING TO THIS DIRECTIVE, EACH WAITING FOR THE OTHER TO PROCEED.

(IN THIS CASE, THERE WERE TWO PROCESSES IN THE CIRCULAR CHAIN. IN GENERAL, PATTERNS LIKE THE FOLLOWING ONE ARE POSSIBLE:

> PROCESS 1 WAITS FOR PROCESS 2, WHILE
> PROCESS 2 WAITS FOR PROCESS 3, WHILE
> PROCESS 3 WAITS FOR PROCESS 1.)

DEADLOCK

- A CIRCULAR CHAIN OF PROCESSES, EACH OF WHICH IS UNABLE TO PROCEED UNTIL THE NEXT PROCESS IN THE CHAIN DOES SOMETHING.



INSTRUCTOR NOTES

- BULLET 2:

A POORLY DESIGNED RUNTIME SYSTEM MAY CONFORM TO THE RULES OF Ada AND STILL BE UNFAIR.

PRIORITIES IN Ada ARE DISCUSSED IN SECTION 2. FOR NOW, USE THE TERM INFORMALLY. TO AVOID STARVATION, HIGH PRIORITIES SHOULD BE RESERVED FOR PROCESSES PERFORMING SHORT, URGENT CHORES AT INFREQUENT INTERVALS.

- THE CARTOON:

AFTER STUDYING THE DEADLOCK PROBLEM DESCRIBED ON THE PREVIOUS SLIDE, SOUTHERN PATHETIC RAILROAD REVISED ITS DIRECTIVE AS FOLLOWS:

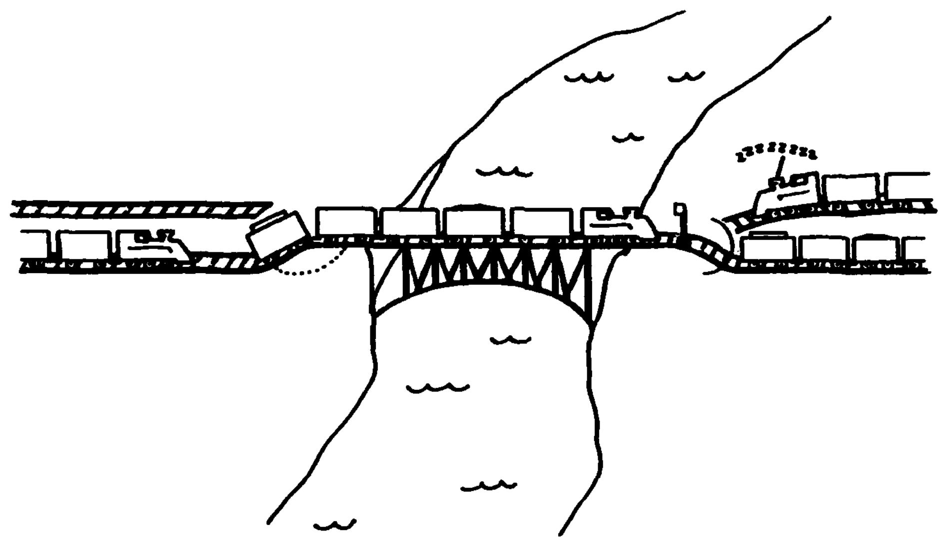
> "UPON APPROACHING THE SHARED TRACK FROM THE WEST, AN EASTBOUND TRAIN SHALL WAIT FOR ANY WESTBOUND TRAIN <u>ALREADY USING</u> THE SHARED TRACK TO PASS, AND THEN PROCEED.
>
> "UPON APPROACHING THE SHARED TRACK FROM THE EAST, A WESTBOUND ENGINEER SHALL TAKE OUT HIS TELESCOPE AND WAIT FOR ANY EASTBOUND TRAIN <u>APPROACHING</u> THE SHARED TRACK TO PASS BEFORE PROCEEDING."

IN OTHER WORDS, EASTBOUND TRAINS HAVE PRIORITY OVER WESTBOUND TRAINS WHEN THERE IS CONTENTION FOR USE OF THE TRACK.

THIS SLIDE DEPICTS A WESTBOUND TRAIN WHOSE ENGINEER HAS FALLEN ASLEEP WHILE WAITING FOR A TIME AT WHICH NO EASTBOUND TRAIN IS APPROACHING. THE WESTBOUND TRAIN NEVER GETS A TURN TO USE THE SHARED TRACK.

STARVATION

- CERTAIN PROCESSES NEVER GETTING A FAIR CHANCE TO EXECUTE.
- POSSIBLE CAUSES:
  - AN UNFAIR RUNTIME SYSTEM
  - HIGH-PRIORITY PROCESSES MONOPOLIZING THE PROCESSOR
  - PROGRAMMING



INSTRUCTOR NOTES

- BULLET 2:

  SYNCHRONIZATION MEANS FORCING ACTIONS PERFORMED BY DIFFERENT PROCESSES TO OCCUR IN A SPECIFIED ORDER.

  COMMUNICATION MEANS LETTING ONE PROCESS USE DATA PRODUCED BY ANOTHER PROCESS.

  IN Ada, SYNCHRONIZATION AND COMMUNICATION ARE BOTH ACHIEVED BY RENDEZVOUS, DESCRIBED IN SECTION 2.

- CARTOON:

  AFTER THE WESTBOUND RIDERS ASSOCIATION TIRED OF HALTING (W.R.A.T.H.) SUED SOUTHERN PATHETIC RAILROAD, THE RAILROAD ADOPTED A NEW SCHEME.

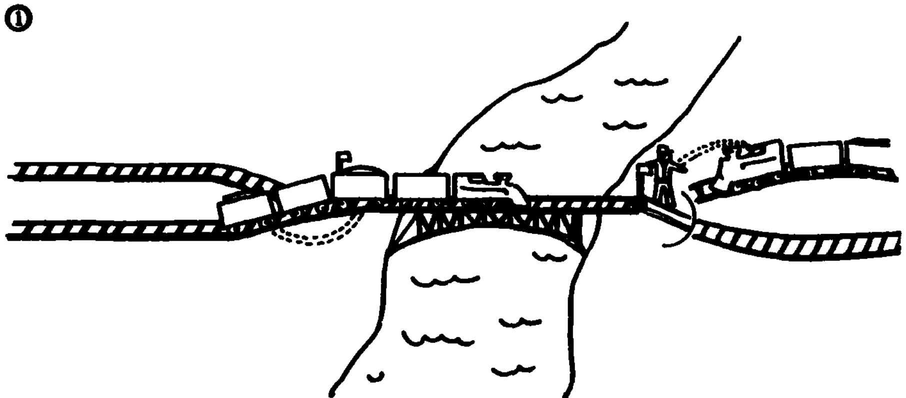
  TRAINS IN EACH DIRECTION NOW HAVE AN EQUAL CHANCE OF GETTING TO USE THE SHARED TRACK FIRST. THERE IS STILL NO CHANCE OF COLLISION, AND VIRTUALLY NO CHANCE OF PERMANENT DEADLOCK (THOUGH IT MAY TAKE A WHILE TO DETERMINE WHICH TRAIN GOES FIRST).

  THE RAILROAD NOW RUNS SAFELY, EFFICIENTLY, AND FAIRLY.

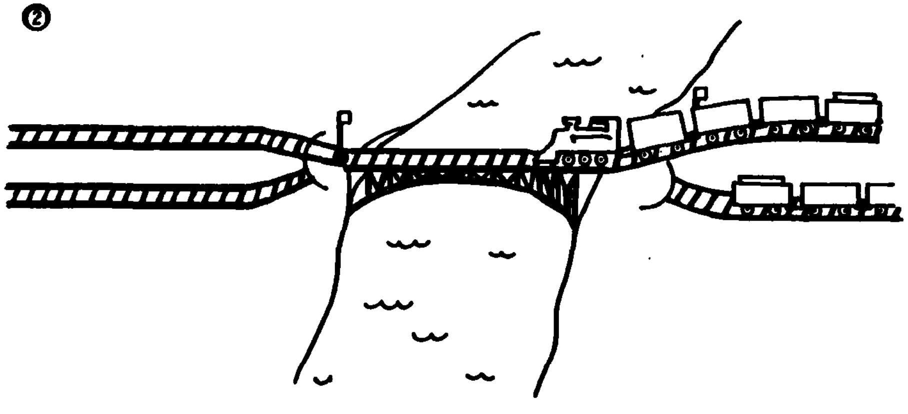
PROCESS COOPERATION

- SIMULTANEOUS UPDATE, DEADLOCK, AND STARVATION WOULD BE NO PROBLEM IF PROCESSES WORKED IN IGNORANCE OF EACH OTHER.
- BECAUSE PROCESSES MUST WORK IN COOPERATION, THEY MUST SYNCHRONIZE AND COMMUNICATE.

①



②



INSTRUCTOR NOTES

- ALLOW 120 MINUTES FOR THIS SECTION.

- THIS SECTION CONSISTS OF THREE LOGICAL SUBSECTIONS

  - TASK TYPES AND TASK OBJECTS (40 MINUTES)

  - TASK COOPERATION (60 MINUTES)

  - SURVEY OF OTHER TASKING FEATURES (20 MINUTES)

THE BEGINNING OF EACH SUBSECTION WILL BE MARKED IN THE INSTRUCTOR NOTES. (THE SUBSECTION INFORMATION IS ONLY PROVIDED TO HELP THE INSTRUCTOR PACE THE COURSE.)

SECTION 2

ADA TASKING FEATURES

VG 831

INSTRUCTOR NOTES

- THIS SLIDE BEGINS THE LOGICAL SUBSECTION - <u>TASKS TYPES AND TASK OBJECTS</u>. ALLOW 45 MINUTES.

- IN THIS SECTION, WE START TO INTRODUCE THE STUDENT TO THE Ada VIEW OF A PROCESS. THE VIEW THAT WE GIVE HERE IS THAT A TASK OBJECT IS A DATA OBJECT THAT BELONGS TO A TASK TYPE. IN THIS RESPECT, AN Ada TASK OBJECT IS NOT MUCH DIFFERENT FROM DATA OBJECTS WE TALKED ABOUT IN EARLIER COURSES. THIS IS THE MAIN POINT OF THIS SECTION AND IT WILL PERMEATE THE REST OF THE COURSE. EVERY ATTEMPT SHOULD BE MADE TO GET THIS POINT ACROSS IN THIS SECTION. HOWEVER, THIS MIGHT BE TOO TASKING (SORRY ABOUT THAT!) FOR MOST OF THE STUDENTS TO GRASP RIGHT AWAY.

WHAT IS A PROCESS IN Ada?

- IN Ada, PROCESSES ARE REALIZED AS TASKS.

- EACH PROCESS IS EXECUTED BY ITS OWN VIRTUAL PROCESSOR.

- A <u>TASK OBJECT</u> IS A DATA OBJECT CONSISTING OF:

  - A PROCESS EXECUTING A PARTICULAR SEQUENCE OF STATEMENTS.
  - A SET OF DATA RESERVED FOR THE EXCLUSIVE USE OF THAT PROCESS.
  - <u>ENTRIES</u> THROUGH WHICH IT IS POSSIBLE TO COMMUNICATE WITH THIS PROCESS.

- LIKE ANY DATA OBJECT IN Ada, A TASK OBJECT BELONGS TO A TYPE - A <u>TASK TYPE</u>.

  - A TASK TYPE CONSISTS OF A SET OF VALUES AND A SET OF OPERATIONS ON THESE VALUES.
  - THE VALUES ARE TASK OBJECTS.
  - OPERATIONS INCLUDE COMMUNICATING WITH A TASK OBJECT THROUGH ONE OF ITS ENTRIES.

- TASK OBJECTS CAN BE

  - COMPONENTS OF RECORDS

```
type Workstation_Type is
   record
      Workstation_Name : String (1 .. 10);
      Terminal_Handler : Terminal_Handler_Task_Type;
   end record;
```

  - COMPONENTS OF ARRAYS

```
type Workstation_Cluster_Type is array (1 .. Max_Workstations) of Workstation_Type;
```

  - POINTED TO BY ACCESS VALUES

```
type Remote_Workstation_Pointer is access Workstation_Type;
```

INSTRUCTOR NOTES

- STARTING WITH THIS SLIDE, WE SHOW THE STUDENTS HOW SOME OF THE CONCEPTS WE HAVE TALKED ABOUT SO FAR, ARE REALIZED IN Ada. WE START BY SHOWING THE BASIC SYNTAX FOR TASK UNITS.

- THE $\boxed{\text{task type name}}$ IS AN IDENTIFIER NAMING THE TASK. WHILE IT IS OPTIONAL AT THE END OF THE TASK TYPE DECLARATION AND TASK BODY, GOOD PROGRAMMING STYLE SUGGESTS THAT IT NEVER BE OMITTED.

- MENTION THAT ENTRY DECLARATIONS ARE LIKE PROCEDURE DECLARATIONS (REPLACE <u>procedure</u> WITH <u>entry</u>). PARAMETER MODES <u>in</u>, <u>in out</u> AND <u>out</u> ARE ALLOWED AS ARE DEFAULT INITIAL VALUES FOR <u>in</u> PARAMETERS.

- IN THE TASK BODY $\boxed{\text{sequence of statements}}$ DEFINES THE SEQUENCE OF STATEMENTS THAT A TASK IN THIS TASK TYPE WILL EXECUTE.

- THE TASK BODY MAY CONTAIN CERTAIN OTHER STATEMENTS NOT ALLOWED ELSEWHERE IN Ada PROGRAMS.

DECLARING A TASK TYPE

- THE DEFINITION OF A TASK TYPE HAS <u>TWO</u> PARTS:
  - A TASK TYPE DECLARATION
    - DECLARES THE TASK TYPE.
    - DECLARES THE FORM OF ENTRIES FOR TASKS IN THE TASK TYPE.

```
task type task type name is
  {entry entry name [(formal parameters)];}
end [task type name];
```

  - A TASK BODY
    - DEFINES A TEMPLATE FOR DATA USED BY EACH TASK IN THE TYPE.
    - DEFINES THE SEQUENCE OF STATEMENTS TO BE EXECUTED BY EACH TASK IN THE TYPE.

```
task body task type name is
   declarative part
begin
   sequence of statements
   [exception
      sequence of exception handlers]
end [task type name];
```

- EACH TASK IN THE TASK TYPE
  - HAS ENTRIES OF THE FORM DESCRIBED IN THE TASK TYPE DECLARATION.
  - HAS ITS OWN PRIVATE COPY OF THE DATA DESCRIBED BY THE TEMPLATE IN THE TASK BODY.
  - EXECUTES THE SEQUENCE OF STATEMENTS IN THE TASK BODY (AT A SPEED INDEPENDENT OF ANY OTHER TASK OF THE SAME TYPE).
- THE TASK TYPE DECLARATION AND TASK BODY TOGETHER ARE CALLED A <u>TASK</u> <u>UNIT</u>.

INSTRUCTOR NOTES

- FOR THIS SLIDE:
  - POINT OUT THE TASK DECLARATION
    - POINT OUT THE ENTRY DECLARATIONS
    - POINT OUT HOW MUCH THEY LOOK LIKE PROCEDURE DECLARATIONS
    - POINT OUT THE PARAMETER MODES
  - POINT OUT THE TASK BODY
    - POINT OUT THE DECLARATIVE PART
    - POINT OUT THE SEQUENCE OF STATEMENTS AND MENTION THAT THERE ARE STATEMENTS (SELECT, ACCEPT) THAT WILL BE EXPLAINED LATER

EXAMPLE OF A TASK UNIT DECLARATION

TASK TYPE DECLARATION

```
task type Shared_Count_Type is

   entry Increase_Count (By : in Positive);
   entry Get_Count (Sum_So_Far : out Natural);

end Shared_Count_Type;
```

TASK BODY

```
task body Shared_Count_Type is

   Sum : Natural := 0;

begin -- Shared_Count_Type

   loop

      select

         accept Increase_Count (By : in Positive) do
            Sum := Sum + By;
         end Increase_Count;

      or

         accept Get_Count (Sum_So_Far : out Natural) do
            Sum_So_Far := Sum;
         end Get_Count;

      end select;

   end loop;

end Shared_Count_Type;
```

INSTRUCTOR NOTES

A TASK TYPE DECLARATION CAN GO IN THE DECLARATIVE PART OF A PROCEDURE BODY, FUNCTION BODY, OUTER TASK BODY, OR PACKAGE BODY, WITH THE CORRESPONDING TASK BODY FOLLOWING LATER IN THE SAME DECLARATIVE PART. ALTERNATIVELY, THE TASK TYPE DECLARATION CAN GO IN A PACKAGE DECLARATION AND THE TASK BODY IN THE CORRESPONDING PACKAGE BODY.

THE UPPER RIGHT EXAMPLE ILLUSTRATES NESTED TASK BODIES.

IN THE LOWER LEFT EXAMPLE, THE TASK TYPE IS PART OF THE PACKAGE IMPLEMENTATION, AND IS HIDDEN FROM THE USER OF THE PACKAGE. IN THE LOWER RIGHT EXAMPLE, THE TASK TYPE IS PROVIDED TO USERS OF THE PACKAGE.

WHERE TASK UNITS GO

```
procedure P is
   ...
  task type declaration
   ...
  task body
   ...
begin
   ...
end P;
```

```
function F return T is
   ...
  task type declaration
   ...
  task body
   ...
begin
   ...
end F;
```

```
task body Outer_Task is
   ...
  task type declaration
   ...
  task body
   ...
begin
   ...
end Outer_Task;
```

```
package P is
   ...
end P;

package body P is
   ...
  task type declaration
   ...
  task body
   ...
begin
   ...
end P;
```

```
package P is
   ...
  task type declaration
   ...
end P;

package body P is
   ...
  task body
   ...
begin
   ...
end P;
```

INSTRUCTOR NOTES

- THE PACKAGE HAS A TASK TYPE DECLARATION IN ITS PACKAGE SPECIFICATION AND THE CORRESPONDING TASK BODY IN THE PACKAGE BODY. THIS SHOWS THE CLASS THAT THE TASK TYPE DEFINITION MAY BE USED LIKE ANY Ada TYPE.

- THE EXAMPLE SHOWS HOW A TASK OBJECT IS DECLARED. THIS IS THE FIRST EXAMPLE OF AN ENTRY CALL THAT HAS BEEN SHOWN, SO GO OVER IT IN SOME DETAIL.

- AS BACKGROUND FOR THE EXAMPLE, EACH ROOM IN A NUCLEAR POWER PLANT MIGHT HAVE A GEIGER COUNTER TO MONITOR RADIATION LEVELS. AS THE SENSORS OF THE GEIGER COUNTER SENSE RADIATION, THEY CALL THE Increase_Count ENTRY OF A Shared_Count_Type OBJECT. THIS ALLOWS THE GEIGER COUNTER TO KEEP TRACK OF THE NUMBER OF GEIGERS COUNTED SO FAR. WHEN TOO MANY GEIGERS ARE COUNTED AN ALARM IS SOUNDED.

- <u>WARNING</u>: THE *PREVIOUS EXAMPLE* IS <u>Tongue-In-Cheek</u>. GEIGER COUNTERS DO NOT REALLY COUNT GEIGERS.

- MENTION THAT TASK UNITS CANNOT BE COMPILED SEPARATELY AS LIBRARY UNITS, HOWEVER THIS EXAMPLE SHOWS HOW TO ACHIEVE THE SAME EFFECT.

DECLARING TASK OBJECTS

```
package Shared_Count_Package is

   task type Shared_Count_Type is
      ...
   end Shared_Count_Type;

end Shared_Count_Package;


package body Shared_Count_Package is

   task body Shared_Count_Type is
      ...
   end Shared_Count_Type;

end Shared_Count_Package;


with Shared_Count_Package;
procedure Count_Geigers is
      ...
   Number_Of_Geigers_Counted   : Positive;
   Control_Room_Geiger_Counter : Shared_Count_Package.Shared_Count_Type;
      ...
begin
      ...
   Control_Room_Geiger_Counter.Increase_Count (By => 1);
   Control_Room_Geiger_Counter.Get_Count (Sum_So_Far => Number_Of_Geigers_Counted);
      ...
end Count_Geigers;
```

INSTRUCTOR NOTES

- BULLET 1
  - ITEM 1
    - REMEMBER THAT IN CERTAIN CONTEXT, SUBPROGRAM DECLARATIONS MAY BE OMITTED, AND SUBPROGRAM BODIES WILL SERVE BOTH ROLES. THIS IS NOT SO FOR THE OTHER PROGRAM UNITS.
  - ITEM 4
    - A CALL ON A SUBPROGRAM MAY APPEAR AFTER THE SUBPROGRAM DECLARATION AND BEFORE THE BODY.
    - AN ENTITY PROVIDED BY A PACKAGE MAY BE REFERRED TO AFTER THE PACKAGE DECLARATION AND BEFORE THE PACKAGE BODY.
    - A GENERIC UNIT MAY BE INSTANTIATED AFTER THE GENERIC DECLARATION AND BEFORE THE GENERIC BODY.
    - A TASK TYPE MAY BE NAMED IN OBJECT DECLARATIONS, TYPE DECLARATIONS ETC., AFTER THE TASK TYPE DECLARATION AND BEFORE THE TASK BODY. SIMILARLY, ENTRY CALLS FOR OBJECTS OF THE TYPE MAY APPEAR AFTER THE TASK TYPE DECLARATION AND BEFORE THE TASK BODY.
- MAKE SURE THAT THE CLASS REALIZES THAT THE DIFFERENCES DO NOT RESULT IN ANY REAL LIMITATIONS.

PROGRAM UNITS

	SUBPROGRAMS	PACKAGES	GENERIC UNITS	TASK UNITS
EXTERNAL VIEW	SUBPROGRAM DECLARATION	PACKAGE DECLARATION	GENERIC SUBPROGRAM OR GENERIC PACKAGE DECLARATION	TASK TYPE DECLARATION
IMPLEMENTATION	SUBPROGRAM BODY	PACKAGE BODY	SUBPROGRAM OR PACKAGE BODY	TASK BODY

- SIMILARITIES AMONG PROGRAM UNITS
  - EACH HAS TWO PARTS: A DECLARATION DESCRIBING THE EXTERNAL VIEW AND A BODY DESCRIBING THE IMPLEMENTATION.
  - THE EXTERNAL VIEW AND BODY MAY BOTH APPEAR IN A DECLARATIVE PART.
  - THE EXTERNAL VIEW CAN BE GIVEN IN A PACKAGE DECLARATION WITH THE INTERNAL VIEW IN A PACKAGE BODY.
  - EXTERNAL VIEW IS SUFFICIENT TO ALLOW USE OF THE UNIT.

- SEPARATE COMPILATION
  - FOR SUBPROGRAMS, PACKAGES AND GENERIC UNITS:
    - THE EXTERNAL VIEW MAY BE COMPILED SEPARATELY AS A LIBRARY UNIT (TO BE MADE AVAILABLE THROUGH A with CLAUSE), AND
    - THE INTERNAL VIEW CAN BE COMPILED LATER AS A SECONDARY UNIT.
  - SEPARATE COMPILATION OF TASK UNITS IS ACCOMPLISHED BY ENCLOSING THE TASK UNIT IN A PACKAGE.

- GENERIC UNITS
  - THERE ARE GENERIC SUBPROGRAMS AND GENERIC PACKAGES.
  - THE EFFECT OF A "GENERIC TASK" IS ACHIEVED BY ENCLOSING A TASK UNIT IN A GENERIC PACKAGE.

INSTRUCTOR NOTES

- THIS SLIDE SERVES AS PREPARATION FOR THE NEXT SLIDE ON ANONYMOUS TASKS, BY REMINDING THE STUDENTS THAT THEY HAVE SEEN ANONYMOUS TYPES BEFORE.

- REMIND STUDENTS THAT WE OCCASIONALLY WANT TO DEFINE ONE-OF-A-KIND ARRAYS.

REVIEW - ANONYMOUS ARRAY TYPES

- Ada PROVIDES A SHORTHAND FOR DECLARING "ONE-OF-A-KIND" ARRAYS.

- THE DECLARATIONS

```
type Days_In_Month_Type is array (Month_Type) of Positive;
Days_In_Month : Days_In_Month_Type;
```

CAN BE ABBREVIATED BY A SPECIAL KIND OF OBJECT DECLARATION:

```
Days_In_Month : array (Month_Type) of Positive;
```

- THE OBJECT Days_In_Month IS SAID TO BELONG TO AN <u>ANONYMOUS ARRAY TYPE</u>.

INSTRUCTOR NOTES

- JUST AS WE SOMETIMES WANT TO DEFINE ONE-OF-A-KIND ARRAYS, WE SOMETIMES WANT TO DEFINE ONE-OF-A-KIND TASKS. AS AN EXAMPLE, IF WE ONLY WANTED ONE MESSAGE BUFFER FOR MESSAGES OF TYPE Message_Type THEN THE SLIDE SHOWS HOW TO DECLARE SUCH A TASK.

- THE SECOND VERSION IS IDENTICAL TO THE FIRST EXCEPT THAT THE RESERVED WORD type HAS BEEN OMITTED. THIS IS THE ONLY SYNTACTIC DIFFERENCE FROM THE FIRST ONE. OF COURSE, THE IDENTIFIER Message_Buffer IS THE NAME OF A TASK OBJECT, NOT A TASK TYPE. Ada TREATS THIS TASK AS BELONGING TO AN ANONYMOUS TASK TYPE JUST AS IN THE CASE OF ANONYMOUS ARRAYS. IN CASE A STUDENT ASKS HOW THIS IS DONE, JUST SAY THAT IT IS TREATED AS IF THE FOLLOWING DECLARATION EXISTED:

```
task type Anonymous_Type_For_Message_Buffer is ...;
Message_Buffer : Anonymous_Type_For_Message_Buffer;
```

ANONYMOUS TASK TYPES

- <u>THE SEQUENCE OF DECLARATIONS</u>:

```
task type Message_Buffer_Type is
   ...
end Message_Buffer_Type;
task body Message_Buffer_Type is
   ...
end Message_Buffer_Type;
Message_Buffer : Message_Buffer_Type;
```

<u>CAN BE ABBREVIATED AS</u>:

```
task Message_Buffer is  -- THE WORD type IS MISSING AFTER task
   ...
end  Message_Buffer;
task body Message_Buffer is
   ...
end Message_Buffer;
```

INSTRUCTOR NOTES

- SOMETIMES WE WANT TO HAVE A TASK THAT INITIATES COMMUNICATION WITH OTHER TASKS, BUT DOES NOT ITSELF NEED TO HAVE OTHER TASKS INITIATE CONVERSATION WITH IT. IN THIS CASE, WE HAVE A TASK WITHOUT ENTRIES. EXAMPLES WILL BE GIVEN LATER IN THE COURSE.

- BULLET #1 - THIS SHOWS THE TASK TYPE DECLARATION FOR A TASK TYPE WITHOUT ENTRIES.

- BULLET #2 - THIS SHOWS THE TASK DECLARATION FOR A ONE-OF-A-KIND TASK WITHOUT ENTRIES.

TASKS WITHOUT ENTRIES

- A TASK TYPE DECLARATION FOR TASKS WITHOUT ENTRIES:

  ```
  task type [task type name] is
  end [task type name];
  ```

  CAN BE ABBREVIATED AS:

  ```
  task type [task type name];
  ```

- A TASK DECLARATION FOR ONE-OF-A-KIND TASK OBJECTS WITHOUT ENTRIES

  ```
  task [task object name] is
  end [task object name];
  ```

  CAN BE ABBREVIATED AS:

  ```
  task [task object name];
  ```

- EXAMPLES:

  ```
  task type Alarm_Task_Type;
  task Control_Panel_Task;
  ```

INSTRUCTOR NOTES

THE PURPOSE OF THIS SLIDE IS TO AVOID CONFUSION BETWEEN THE CYCLIC EXECUTIVE NOTION OF TASK INITIATION AND THE Ada NOTION OF TASK ACTIVATION.

Ada PROVIDES OTHER WAYS TO

- CONTROL THE INTERVALS AT WHICH THE PROCESSING LOOP IS REPEATED. (CYCLIC PROCESSING IS COVERED IN SECTION 3.)
- DYNAMICALLY ACTIVATE AND TERMINATE TASKS WHEN THAT ABILITY IS REQUIRED FOR SOME OTHER REASON. (DYNAMIC ACTIVATION BY ALLOCATING TASKS IS DISCUSSED IN THIS SECTION. DYNAMIC TERMINATION, BECAUSE OF A CALL ON A PARTICULAR ENTRY OR THROUGH AN ABORT STATEMENT, IS DISCUSSED LATER.)

- BULLET #3
  - THE VIEW DESCRIBED HERE IS USUALLY MORE APPROPRIATE BECAUSE IT ALLOWS EACH TASK TO PROCESS A SINGLE CONCEPTUAL THREAD.

TASK ACTIVATION AND TERMINATION

- WHEN DEALING WITH TASKS WE NEED TO KNOW:
  - WHEN AND HOW ARE TASKS STARTED UP? - ACTIVATION
  - WHEN AND HOW DO TASKS FINISH? - TERMINATION

- TRADITIONAL VIEW (WITH CYCLIC EXECUTIVES):
  - A TASK IS "INITIATED" EACH TIME ITS TURN ARRIVES.
  - EACH INITIATION DOES THE WORK REQUIRED OF THE TASK ON ONE DUTY CYCLE, AND THEN TERMINATES.

- IN Ada PROGRAMS, ANOTHER VIEW IS USUALLY MORE APPROPRIATE:
  - A TASK FOR PERFORMING PERIODIC PROCESSING IS ACTIVATED ONCE, AT THE BEGINNING OF THE PROGRAM.
  - THE TASK EXECUTES A LOOP THAT IS REPEATED ONCE FOR EACH PROCESSING PERIOD.

INSTRUCTOR NOTES

THIS OVERVIEW OMITS THE FOLLOWING DETAILS:

- FOR A TASK OBJECT OR ACCESS TYPE DECLARED IN A NON-LIBRARY PACKAGE, THE SAME RULES APPLY AS IF IT WERE DECLARED JUST AFTER THE PACKAGE.

- FOR A TASK OBJECT OR ACCESS TYPE DECLARED IN A LIBRARY PACKAGE, NO WAITING IS EVER REQUIRED. THERE IS NO SUCH THING AS "DEPARTURE FROM A LIBRARY PACKAGE."

- SIMILAR RULES APPLY TO DECLARED RECORDS AND ARRAYS WITH TASK OBJECT COMPONENTS AND TO ACCESS TYPES POINTING TO SUCH RECORDS AND ARRAYS.

TERMINATION OF TASKS

- WHEN A TASK OBJECT IS DECLARED IN THE DECLARATION PART OF A SUBPROGRAM, BLOCK STATEMENT, OR OUTER TASK BODY:
  - DEPARTURE FROM THE SUBPROGRAM, BLOCK STATEMENT, OR OUTER TASK CANNOT TAKE PLACE UNTIL THE DECLARED TASK TERMINATES:

```
function Sorted_List (List : List_Type) return List_Type is
   Left_Half_Sorter, Right_Half_Sorter : Sorting_Task_Type;
      -- two declared task objects
   ...
begin
   ...
   return Result;  <-- FUNCTION WAITS HERE UNTIL BOTH
                       Left_Half_Sorter AND Right_Half_Sorter
                       HAVE TERMINATED
   ...
end Sorted_List;
```

- IF AN ACCESS TYPE POINTING TO TASK OBJECTS IS DECLARED IN THE DECLARATIVE PART OF A SUBPROGRAM, BLOCK STATEMENT, OR OUTER TASK BODY:
  - DEPARTURE FROM THE SUBPROGRAM, BLOCK STATEMENT, OR OUTER FRAME CANNOT TAKE PLACE UNTIL ALL TASKS POINTED TO BY VALUES IN THAT TYPE HAVE TERMINATED.

INSTRUCTOR NOTES

THIS OVERVIEW OMITS THE FOLLOWING DETAILS:

- IF A TASK OBJECT IS DECLARED IN A NON-LIBRARY PACKAGE, THE EFFECT IS AS IF THE TASK OBJECT WERE DECLARED IN THE SAME DECLARATIVE PART AS THE PACKAGE ITSELF.

- IF A TASK OBJECT IS A COMPONENT OF AN ARRAY OR RECORD, THE SAME RULES APPLY, BASED ON THE DECLARATION OR ALLOCATION OF THE ARRAY OR RECORD.

- IF A PACKAGE BODY HAS NO SEQUENCE OF STATEMENTS, IT IS ACTIVATED JUST AFTER THE LAST DECLARATION IN THE BODY IS ELABORATED.

- BULLET 1:
  - ITEM 2: THIS IS BECAUSE THE SEQUENCE OF STATEMENTS IN A LIBRARY PACKAGE'S BODY IS EXECUTED BEFORE THE MAIN PROGRAM BEGINS EXECUTION.

ACTIVATION OF TASKS

- WHEN A TASK OBJECT IS DECLARED IN THE DECLARATIVE PART OF A SUBPROGRAM, BLOCK STATEMENT, OR TASK BODY:

  - THE TASK BEGINS EXECUTION JUST AS THE CORRESPONDING SEQUENCE OF STATEMENTS IS ENTERED:

```
procedure Count_Pulses is
   ...
   Pulse_Count : Shared_Count_Type;   -- declared task object
   ...
begin
         Pulse_Count begins execution here.

end Count_Pulses;
```

  - SPECIAL CASE: IF THE TASK OBJECT IS DECLARED IN A LIBRARY PACKAGE, THE TASK BEGINS EXECUTION BEFORE THE MAIN PROGRAM.

- WHEN A TASK OBJECT IS CREATED BY EVALUATION OF AN ALLOCATOR, IT BEGINS EXECUTION UPON ALLOCATION:

```
procedure Count_Pulses is
   type Shared_Count_Pointer_Type is access Shared_Count_Type;
   Pulse_Count_Pointer : Shared_Count_Pointer_Type;
   ...
begin        Pulse_Count_Pointer.all begins execution here.
   ...
   Pulse_Count_Pointer := new Shared_Count_Type;  -- Allocate a new task object and
                                                  -- have Pulse_Count_Pointer
                                                  -- point to it
   ...
end Count_Pulses;
```

INSTRUCTOR NOTES

THIS SLIDE BEGINS THE LOGICAL SUBSECTION ON TASK COOPERATION. ALLOW 45 MINUTES.

USE THE EXAMPLE JUST TO EXPLAIN TERMS LIKE ENTRY, ENTRY CALL, AND accept STATEMENT. DO NOT TRACE THROUGH A RENDEZVOUS UNTIL AFTER THE NEXT SLIDE.

RENDEZVOUS

- TASKS COMMUNICATE THROUGH RENDEZVOUS.
- A RENDEZVOUS OCCURS WHEN ONE TASK CALLS AN ENTRY OF ANOTHER TASK AND THE SECOND TASK EXECUTES AN accept STATEMENT FOR THAT ENTRY.
  - AN ENTRY CALL IS SIMILAR TO A PROCEDURE CALL
  - AN accept STATEMENT PLAYS A ROLE SIMILAR TO A PROCEDURE BODY, BUT IT IS EXECUTED IN TURN WHEN THE CALLED TASK REACHES THAT STATEMENT.

```
procedure Monitor_Volume is
   Volume   : Integer := 0;
   Addition : Integer := 1;
   ...
begin
   ...
   while Volume < 1E3 loop
      ...
      Accumulator_Task.Add_Increment
         (Addition, Volume);
      ...
   end loop;
   ...
end Monitor_Volume
```

entry call

entry declaration

```
task Accumulator_Task is
   entry Add_Increment
      (Increment : in Integer; Sum_So_Far : out Integer);
end Accumulator_Task;

task body Accumulator_Task is
   Sum : Integer := 0;
begin
   loop
      accept Add_Increment
         (Increment : in Integer;
          Sum_So_Far : out Integer) do
         Sum := Sum + Increment;
         Sum_So_Far := Sum;
      end Add_Increment;
   end loop;
end Accumulator_Task;
```

accept statement

INSTRUCTOR NOTES

AFTER EXPLAINING THESE FIVE STEPS, GO BACK TO THE PREVIOUS SLIDE AND USE THE EXAMPLE TO ILLUSTRATE EACH STEP.

EVENTS IN A RENDEZVOUS

1. WHICHEVER TASK IS READY FIRST WAITS FOR THE OTHER.
   - IF THE CALLING TASK ISSUES AN ENTRY CALL FIRST, IT WAITS FOR THE CALLED TASK TO ARRIVE AT AN accept STATEMENT FOR THAT ENTRY
   - IF THE CALLED TASK REACHES AN accept STATEMENT FOR SOME ENTRY FIRST, IT WAITS FOR SOME TASK TO CALL THAT ENTRY

2. WHEN BOTH TASKS ARE READY, PARAMETERS OF MODE in OR in out ARE COPIED FROM THE CALLING TASK TO THE CALLED TASK.

3. THE STATEMENTS INSIDE THE accept STATEMENT ARE EXECUTED.

4. PARAMETERS OF MODE in out OR out ARE COPIED FROM THE CALLED TASK BACK TO THE CALLING TASK.

5. THE CALLING TASK RESUMES EXECUTION JUST AFTER THE ENTRY CALL AND THE CALLED TASK RESUMES EXECUTION JUST AFTER THE accept STATEMENT.

INSTRUCTOR NOTES

- THIS IS INTRODUCED AS A SIMPLE CASE OF A COMPLETE TASK TYPE.

- THIS SLIDE REVISITS SIMULTANEOUS UPDATE, SO BRIEFLY REVIEW THE PROBLEM.

- THE LOCK SOLUTION IS GOOD AS AN EXAMPLE BUT THE MONITOR SOLUTION IN SECTION 3 IS A BETTER WAY TO ACHIEVE MUTUAL EXCLUSION.

- MAKE SURE THE STUDENTS UNDERSTAND HOW THIS EXAMPLE SOLVES THE SIMULTANEOUS UPDATE PROBLEM.

- NOTICE THAT THE ACCEPT STATEMENT DOES NOT HAVE A SEQUENCE OF STATEMENTS. IN PARTICULAR

```
accept Lock;
```

  IS AN ABBREVIATION FOR

```
accept Lock do
   null;
end Lock;
```

SIMULTANEOUS UPDATE USING LOCKS - A PRIMITIVE SOLUTION

- THE PROBLEM OF SIMULTANEOUS UPDATE WAS DESCRIBED IN SECTION 1. A PRIMITIVE SOLUTION USES LOCKS.

- EXAMPLE OF THE USE OF LOCKS:

```
Database_Lock : Lock_Type;  -- Lock_Type IS A TASK TYPE WITH
                            -- ENTRIES Lock AND Unlock.
   ...
Database_Lock.Lock;
-- THE CALLING TASK NOW HAS EXCLUSIVE USE OF THE
-- DATABASE.  OTHER TASKS CALLING Database_Lock.Lock
-- MUST WAIT.
Database_Lock.Unlock;
-- NOW OTHER CALLS ON Database_Lock.Lock CAN BE
-- ACCEPTED.
```

- IMPLEMENTATION OF Lock_Type:

```
task type Lock_Type is

   entry Lock;
   entry Unlock;

end Lock_Type;

task body Lock_Type is
begin

   loop
      accept Lock;
      -- THE Lock_Type TASK NOW WAITS FOR A CALL ON Unlock.
      -- ANOTHER TASK CALLING Lock AT THIS POINT WILL
      -- HAVE TO WAIT UNTIL Unlock HAS BEEN CALLED
      -- SO THIS TASK CAN AGAIN ACCEPT Lock.
      accept Unlock;
   end loop;
end Lock_Type;
```

- A MORE ABSTRACT AND LESS ERROR-PRONE SOLUTION USES MONITORS, WHICH WILL BE INTRODUCED LATER.

INSTRUCTOR NOTES

- THIS SLIDE MOTIVATES THE NEED FOR A SIMPLE SELECTIVE WAIT.

PROBLEM-SELECTING ONE OF SEVERAL ENTRY CALLS

- IN THE Lock_Type TASK
  - Lock MUST BE CALLED FIRST
  - Unlock IS CALLED AFTER Lock

  THIS YIELDS THE PREDICTABLE SEQUENCE

  Lock, Unlock, Lock, Unlock

- MANY TIMES THE CALLING SEQUENCE IS NOT PREDICTABLE.
  - THE Shared_Count_Type HAS TWO ENTRIES
    - Increase_Count
    - Get_Count
  - THESE ENTRIES CAN BE CALLED IN ANY ORDER

- THE <u>SELECTIVE WAIT</u> STATEMENT PROVIDES THE NEEDED CAPABILITY.

INSTRUCTOR NOTES

- THIS SLIDE PRESENTS THE TASK BODY OF THE Shared_Count_Type DISCUSSED EARLIER IN THE COURSE. IT SHOWS HOW THE SIMPLE SELECTIVE WAIT WORKS.

- EXPLAIN HOW THE SELECT STATEMENT WORKS. (POINT OUT THAT Get_Count CAN BE CALLED BEFORE Increase_Count SINCE Sum HAS AN INITIAL VALUE.)

- IN EXECUTING THE SELECTIVE WAIT, ONE OF TWO CASES CAN OCCUR:

  1. NONE OF THE ENTRIES HAS BEEN CALLED:
     - WAIT UNTIL ONE IS CALLED AND ACCEPT THAT CALL.
  2. ONE OR MORE OF THE ENTRIES HAS BEEN CALLED:
     - ONE ACCEPT STATEMENT THAT CAN BE EXECUTED IMMEDIATELY IS SELECTED <u>ARBITRARILY AND EXECUTED</u>.

  THE METHOD OF ARBITRARY SELECTION DEPENDS ON THE RUNTIME SYSTEM, BUT A GOOD RUNTIME SYSTEM WILL BE FAIR. THE PROGRAMMER SHOULD VIEW THE CHOICE AS RANDOM.

- MAKE SURE YOU INTRODUCE THE TERM "SELECTIVE WAIT ALTERNATIVE."

EXAMPLE OF A SELECTIVE WAIT

```
task type Shared_Count_Type is
   entry Increase_Count (By : in Positive);
   entry Get_Count (Sum_So_Far : out Natural);
end Shared_Count_Type;
task body Shared_Count_Type is
   Sum : Natural := 0;
begin -- Shared_Count_Type
   loop
     select
        accept Increase_Count (By : in Positive) do      } SELECTIVE WAIT ALTETNATIVE
           Sum := Sum + By;
        end Increase_Count;
     or
        accept Get_Count (Sum_So_Far : out Natural) do   } SELECTIVE WAIT ALTERNATIVE
           Sum_So_Far := Sum;
        end Get_Count;
     end select;
   end loop;
end Shared_Count_Type;
```

- IF ONE OR MORE OF THE ENTRIES HAVE BEEN CALLED ONE IS CHOSEN ARBITRARILY
- OTHERWISE, WAIT FOR A CALL ON ANY ENTRY (WHICHEVER IS CALLED FIRST)

INSTRUCTOR NOTES

- THIS EXAMPLE SHOWS THAT THE ACCEPT STATEMENT CAN HAVE A SEQUENCE OF STATEMENTS AFTER IT. THE STATEMENTS IN A GIVEN SELECTIVE WAIT ALTERNATIVE ARE EXECUTED ONLY AFTER THE CORRESPONDING accept STATEMENT IS SELECTED. IT IS DESIRABLE TO MOVE COMPUTATION OUT OF THE accept STATEMENT AND INTO THE FOLLOWING SEQUENCE OF STATEMENTS SO THAT THE CALLING TASK CAN MOVE ON PAST THE ENTRY CALL STATEMENT.

- THE FOLLOWING LOOP MIGHT BE PART OF AN ONBOARD NAVIGATION SYSTEM.

- A PILOT SETS A DESTINATION WHICH RESULTS IN THE Set_Destination ENTRY BEING CALLED.

- PERIODICALLY, THE CURRENT POSITION IS REPORTED, AT WHICH TIME IT IS COMPARED WITH THE CURRENT DESTINATION. IF THE CURRENT POSITION IS WITHIN RANGE OF THE DESTINATION, THE PILOT IS ALERTED.

- THE PILOT ALSO SETS A PROXIMITY RANGE WHICH RESULTS IN THE Set_Proximity ENTRY BEING CALLED.

- IT SHOULD BE ASSUMED THAT Current_Alert_Range AND Current_Destination HAVE DEFAULT INITIAL VALUES (OR ARE SET BEFORE THE Loop IS ENTERED) IN CASE Report IS CALLED BEFORE THE OTHER TWO ENTRIES.

ANOTHER EXAMPLE

```
loop

   select

      accept Set_Proximity (Alert_Range : in Float) do
         Current_Alert_Range := Alert_Range;
      end Set_Proximity;

   or

      accept Set_Destination (Destination : in Position_Type) do
         Current_Destination := Destination;
      end Set_Destination;

   or

      accept Report (New_Position : in Position_Type) do
         Current_Position := New_Position;
      end Report;
      Distance := Distance_Between (Current_Position, Current_Destination)
      if Distance < Current_Alert_Range then
         Alert_Pilot;
      end if;

   end select;

end loop;
```

INSTRUCTOR NOTES

- THIS SLIDE MOTIVATES THE NEED FOR GUARDS.

PROBLEM - CONDITIONALLY ACCEPTING AN ENTRY CALL

- A TASK MAY NEED TO GUARD AGAINST ACCEPTING A CALL TO A PARTICULAR ENTRY WHEN SOME CONDITION DOES (DOES NOT) EXIST.

- TASKS OFTEN NEED TO SHARE RESOURCES. ONE SUCH RESOURCE MIGHT BE A POOL OF BUFFERS.

- A NATURAL WAY TO IMPLEMENT THE BUFFERS IS THROUGH A TASK TYPE HAVING THE ENTRIES:
  - Request - REQUESTS AN ARBITRARY BUFFER FROM THE POINT.
  - Release - RELEASES THE BUFFER BACK TO THE POOL.

- BUT WHAT IF THERE ARE NO BUFFERS AVAILABLE?
  - AN ENTRY CALL TO Request SHOULD NOT BE ACCEPTED.
  - CALLS TO Request NEED TO BE ACCEPTED CONDITIONALLY, I.E. WE NEED TO <u>GUARD</u> AGAINST Request CALLS BEING ACCEPTED WHEN THERE ARE NO BUFFERS TO GIVE OUT.
  - Ada PROVIDES THIS BY PROVIDING <u>GUARDS</u>.

- THIS IS A COMPLETE TASK TYPE DEFINITION FOR Buffer_Allocation_Type.
- BUFFERS ARE NUMBERED 1 TO Total_Buffers. A Boolean ARRAY Available IS MAINTAINED TO KEEP TRACK OF WHICH BUFFERS ARE AVAILABLE. BUFFER i IS AVAILABLE IF AND ONLY IF Available (i) = True. TO OBTAIN A BUFFER, Request IS CALLED. Available IS SEARCHED TO FIND AN AVAILABLE BUFFER, I.E., SOME BUFFER i SUCH THAT Available (i) = True. WHEN THIS IS FOUND, THE BUFFER IS MARKED UNAVAILABLE, BY SETTING ITS Available ENTRY TO False. TO RELEASE A BUFFER, Release IS CALLED. THIS SETS THE BUFFERS'S Available COMPONENT TO True.
- WE HAVE A VARIABLE, Number_Of_Buffers_Available, THAT KEEPS TRACK OF Buffers THAT HAVE NOT BEEN ALLOCATED. WE USE THIS TO BUILD A GUARD

  Number_Of_Buffers_Available > 0

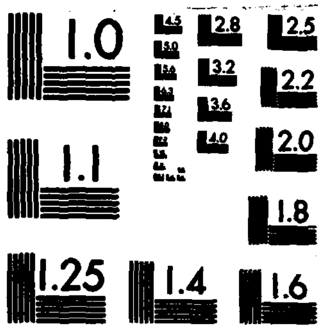
  MEANING "DO NOT SELECT THIS ACCEPT STATEMENT IF THERE ARE NO BUFFERS TO BE GIVEN OUT."
- WHEN Request IS SELECTED, THERE IS ALWAYS AT LEAST ONE BUFFER THAT IS AVAILABLE. THE LOOP WILL ALWAYS TERMINATE VIA THE EXIT STATEMENT, AND A VALUE WILL BE ASSIGNED TO Buffer.
- CONSIDER WHAT WOULD HAPPEN IF THERE WAS NO GUARD AND THE ENTRY IS ACCEPTED WHEN THERE ARE NO BUFFERS. THE LOOP IS NOT LEFT VIA THE exit's STATEMENT, AND NO VALID BUFFER IS ASSIGNED TO Buffer.
- AN ALTERNATIVE IS SAID TO BE <u>OPEN</u> IF
  1. IT HAS NO GUARD, OR
  2. IT HAS A GUARD, AND ITS CONDITION IS TRUE.
- EXECUTION OF A SELECTIVE WAIT STATEMENT
  - FIRST CAUSES ALL GUARDS TO BE EVALUATED,
  - AND THEN PROCEEDS AS WITH THE SIMPLE SELECTIVE WAIT USING ONLY THE OPEN ALTERNATIVES.
- EMPHASIZE THAT IF THE SELECTIVE WAIT STATEMENT MUST WAIT FOR AN ENTRY CALL TO BE MADE, THE <u>GUARDS ARE NOT RE-EVALUATED</u> WHEN A CALL IS MADE.
  - <u>GUARDS ARE ONLY EVALUATED AT T</u>HE START OF THE STATEMENT.
  - A GUARD <u>CANNOT</u> BECOME TRUE WHILE WAITING.

BUFFER ALLOCATION

```
subtype Buffer_Range_Type is range 1 .. Total_Buffers;

task type Buffer_Allocation_Type is
   entry Request (Buffer : out Buffer_Range_Type);
   entry Release (Buffer : in Buffer_Range_Type);
end Buffer_Range_Type;

task body Buffer_Allocation_Type is

   Available                   : array (Buffer_Range_Type) of Boolean := (others =  True);
   Number_Of_Buffers_Available : Natural := Total_Buffers;

begin -- Buffer_Allocation_Type
   loop
      select
        when Number_Of_Buffers_Available > 0 =>
            accept Request (Buffer : out Buffer_Range_Type) do
               Search_Loop:
                  for Candidate_Buffer in Buffer_Range_Type'Range loop
                     if Available (Candidate_Buffer) then
                        Available (Candidate_Buffer) := False;
                        Buffer := Candidate_Buffer;
                        exit Search_Loop;
                     end if;
                  end loop Search_Loop;
               Number_Of_Buffers_Available := Number_Of_Buffers_Available - 1;
            end Request;

      or

            accept Release (Buffer : in Buffer_Range_Type) do
               Available (Buffer) := True;
               Number_Of_Buffers_Available := Number_Of_Buffers_Available + 1;
            end Release;

      end select;
   end loop;
end Buffer_Allocation_Type;
```

AD-A145 093   ADA (TRADEMARK) TRAINING CURRICULUM REAL-TIME CONCEPTS   2/4
L303 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA   JUL 84
DAAB07-83-C-K514

UNCLASSIFIED   F/G 9/2   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

- THIS SLIDE MOTIVATES THE NEED FOR THE terminate ALTERATIVE. THE Shared_Count_Type TASK BODY IS ON SLIDE 2-17.

PROBLEM - TERMINATING A TASK

- THE Shared_Count_Type TASK BODY CONTAINS AN INFINITE LOOP. EACH PASS THROUGH THE LOOP ACCEPTS A CALL ON EITHER THE Increase_Count OR Get_Count ENTRY.
  - THIS TASK CANNOT TERMINATE.
  - ANY SUBPROGRAM, BLOCK OR OUTER TASK BODY CONTAINING A DECLARATION OF A Shared_Count_Type OBJECT CANNOT TERMINATE.

- A <u>TERMINATE ALTERNATIVE</u> IS USED TO SPECIFY THAT A TASK SHOULD TERMINATE WHEN THERE IS NO MORE WORK FOR IT TO DO.

INSTRUCTOR NOTES

- THIS SLIDE DIFFERS FROM THE PREVIOUS Shared_Count_Type SLIDE ONLY BY THE ADDITION OF THE Terminate ALTERNATIVE.

- A PASS THROUGH THE LOOP CAN NOW

  1. ACCEPT AN Increase_Count CALL;
  2. ACCEPT A Get_Count CALL; OR
  3. CAUSE THE TASK TO TERMINATE

     - THIS CAN ONLY HAPPEN IF ITS POTENTIAL CALLERS ARE READY TO TERMINATE BECAUSE THEY ARE

       - WAITING AT THE END OF A BLOCK, SUBPROGRAM, ETC. OR
       - WAITING AT A SELECTIVE WAIT WITH A terminate ALTERNATIVE.

THE terminate ALTERNATIVE

```
task type Shared_Count_Type is

   entry Increase_Count (By : in Positive);
   entry Get_Count (Sum_So_Far : out Natural);

end Shared_Count_Type;

task body Shared_Count_Type is

   Sum : Natural := 0;

begin -- Shared_Count_Type
   loop

      select

         accept Increase_Count (By : in Positive) do
            Sum := Sum + By;
         end Increase_Count;

      or

         accept Get_Count (Sum_So_Far : out Natural) do
            Sum_So_Far := Sum;
         end Get_Count;

      or

         terminate;

      end select;

   end loop;
end Shared_Count_Type;
```

- THE PART OF A PROGRAM THAT CANNOT BE EXITED UNTIL A TASK HAS TERMINATED IS CALLED THE TASK'S MASTER.

- A TASK CAN SELECT A terminate ALTERNATIVE WHEN
  - THE TASK'S MASTER HAS FINISHED EXECUTING ITS SEQUENCE OF STATEMENTS AND DEPARTURE FROM THE MASTER IS AWAITING TERMINATION OF CERTAIN TASKS, AND
  - SELECTION OF terminate ALTERNATIVES WILL CAUSE THESE TASKS TO TERMINATE

- ALL THESE terminate ALTERNATIVES ARE THEN CHOSEN.

INSTRUCTOR NOTES

- THIS SLIDE EXPLAINS WHY TASKS MUST BE ABLE TO DELAY THEMSELVES.
- BULLET 1 - IN ORDER TO TEST A RADAR SYSTEM, ROCKET LAUNCHES ARE SIMULATED. THE TIME BETWEEN LAUNCHES MUST BE AT LEAST SOME FIXED DURATION.
- BULLET 2 - AS A SATELLITE PROGRESSES ALONG ITS TRAJECTORY, IT MAY LOSE CONTACT WITH THE GROUND STATION (IN THE FIGURE, THE LOSS OCCURS AT POINT A). RATHER THAN BUSY WAITING, THE TASK CALCULATES THE TIME UNTIL IT CAN ACQUIRE A SIGNAL FROM THE NEXT GROUND STATION (AT POINT B IN THE FIGURE). THE TASK DELAYS ITSELF UNTIL THEN.

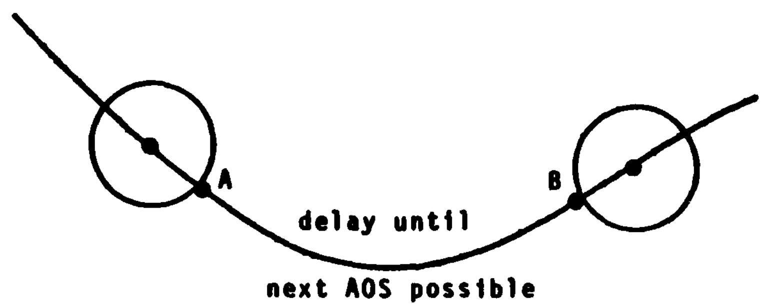
PROBLEM - DELAYING A TASK

- A TASK MIGHT WANT TO DELAY ITSELF BECAUSE IT IS PART OF A REAL-TIME SIMULATION (SUCH AS A DRIVER-TRAINING SIMULATION):

```
Display_Yellow_Light;
-- delay for duration of a yellow light (4 seconds)
Display_Red_Light;
```

- A TASK MIGHT WANT TO DELAY ITSELF TO AVOID <u>BUSY WAITING</u>, SAY, IN A SATELLITE TELEMETRY TRANSMISSION TASK:

```
-- Compute next expected AOS - (Acquisition Of Signal)
-- Delay until then
-- Try to establish communications
```



- A TASK MIGHT WANT TO DELAY ITSELF IF IT IS TO PERFORM SOME ACTIVITY PERIODICALLY.

```
loop
   Sample_Temperature (T);  -- sampled every 200 milliseconds
   -- delay until next cycle
end loop;
```

INSTRUCTOR NOTES

- THE DELAY STATEMENT DELAYS THE TASK FOR AT LEAST THE DURATION SPECIFIED.
  - THE RUNTIME SYSTEM MIGHT NOT ALLOCATE THE CPU TO A TASK THE INSTANT THAT ITS DELAY EXPIRES, SO THE EFFECTIVE DELAY CAN BE LONGER.

- POINT OUT THE USE OF Clock.

- THE EXPRESSION Time_Of_Next_AOS - Calendar.Clock SUBTRACTS TWO OBJECTS OF TYPE Time AND YIELDS AN EXPRESSION OF TYPE Duration.

- THE TYPE Duration HAS THE NORMAL FIXED POINT OPERATIONS.
  - Duration VALUES CAN BE ADDED TO AND SUBTRACTED FROM TIME VALUE.
  - EACH IMPLEMENTATION MUST PROVIDE FOR AT LEAST -86,400 .. 86,400 SECONDS (1 DAY).

- MAKE SURE THE CLASS UNDERSTANDS THE DIFFERENCE BETWEEN Time (A POINT ON A TIME LINE) AND Duration (THE DISTANCE BETWEEN TWO POINTS ON A LINE.)

DELAY STATEMENT

- Duration IS A PREDEFINED FIXED POINT TYPE (FOR AMOUNTS OF TIME)
- THE PREDEFINED PACKAGE Calendar PROVIDES A TYPE TIME (FOR POINTS IN TIME) AND A FUNCTION NAMED Clock THAT RETURNS THE CURRENT TIME
- THE delay STATEMENT DELAYS THE TASK FOR AT LEAST THE SPECIFIED DURATION.

EXAMPLE:

```
       ...
Time_Between_Launches, : Duration;
Next_Launch_Time       : Calendar.Time;
       ...
   Next_Launch_Time := Calendar.Clock;
   for I in 1 .. Number_Of_Rockets loop
      Simulate_Rocket_Launch;
      Next_Launch_Time := Next_Launch_Time + Time_Between_Launches;
      delay Next_Launch_Time - Calendar.Clock;
   end loop;
```

EXAMPLE:

```
       ...
   Time_Of_Next_AOS : Calendar.Time;
       ...
begin
       ...
   Time_Of_Next_AOS := Compute_Next_Expected_AOS;
   delay Time_Of_Next_AOS - Calendar.Clock;
   Establish_Communications;
       ...
end;
```

INSTRUCTOR NOTES

- THIS SLIDE MOTIVATES THE NEED FOR A DELAY ALTERNATIVE.

PROBLEM - WHEN A TASK'S ENTRIES ARE NOT CALLED IN TIME

- A TASK IN A NAVIGATION SYSTEM ACCEPTS AN ENTRY TO REPORT POSITION AND VELOCITY. THE TASK IN TURN DISPLAYS THE CURRENT POSITION.

- WHAT IF A NEW POSITION DOES NOT ARRIVE IN TIME?
  - THE POSITION GETS OLD
  - THE REPORTED POSITION IS MISLEADING

- A NEW POSITION CAN BE CALCULATED BY EXTRAPOLATION IF THE TASK KNOWS IT HAS WAITED TOO LONG.
  - THE DELAY ALTERNATIVE PROVIDES THIS CAPABILITY.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS HOW THE PROBLEM CAN BE SOLVED.

- THE FUNCTION Projected_Position CALCULATES A NEW POSITION VIA EXTRAPOLATION. IT WILL BE USED IF AN UPDATED POSITION IS NOT RECEIVED IN TIME.

- FOR THE INSTRUCTOR'S BENEFIT, THE PROBLEM OF CUMULATIVE DRIFT - DISCUSSED LATER - IS NOT A PROBLEM HERE. CALLS TO Report_Navigation_Data OCCUR MUCH MORE FREQUENTLY THAN EXTRAPOLATION. THE CALLS EFFECTIVELY SYNCHRONIZE THE LOOP. WELL BEFORE WE COULD EVEN NOTICE CUMULATIVE DRIFT, WE WOULD NEED TO HAVE A SITUATION WHERE Report_Navigation_Data WAS NOT BEING CALLED - A FAR MORE SERIOUS PROBLEM.

- AN OPEN DELAY ALTERNATIVE WILL BE SELECTED ONLY IF NO ACCEPT ALTERNATIVE CAN BE SELECTED BEFORE THE SPECIFIED DELAY HAS ELAPSED. IF SEVERAL DELAY ALTERNATIVES ARE OPEN, THE ONE WITH THE SHORTEST DELAY IS SELECTED - TIES ARE RESOLVED ARBITRARILY.

THE DELAY ALTERNATIVE

- A SELECTIVE WAIT ALTERNATIVE MAY BEGIN WITH A delay STATEMENT RATHER THAN AN accept STATEMENT.

- THE delay STATEMENT HAS A DIFFERENT MEANING IN THIS CONTEXT: THE ALTERNATIVE IS SELECTED IF NO ENTRY CALL ARRIVES BEFORE THE SPECIFIED DURATION

```
loop

   select

      accept Report_Navigation_Data
               (Velocity : in Velocity_Type;
                Position : in Position_Type) do

         Current_Velocity := Velocity;
         Current_Position := Position;

      end Report_Navigation_Data;

   or

      delay 0.5;
      Current_Position := Projected_Position
                            (Current_Position,
                             Current_Velocity);

   end select;

   Display_Position (Current_Position);

end loop;
```

- delay ALTERNATIVES AND terminate ALTERNATIVES ARE MUTUALLY EXCLUSIVE.

INSTRUCTOR NOTES

- THIS SLIDE MOTIVATES THE NEED FOR ELSE PARTS.

PROBLEM - ACCEPTING URGENT ENTRY CALLS

- SOMETIMES A TASK WILL WANT TO REPETITIVELY PERFORM SOME ROUTINE WORK UNLESS SOME EXTRAORDINARY EVENT OCCURS.

- A TASK FOR TRAFFIC LIGHTS MIGHT CHANGE LIGHTS PERIODICALLY.

- WHEN AN EMERGENCY VEHICLE (FIRE, POLICE, ETC.) NEEDS TO PROCEED ALONG THE ROAD WITHOUT INTERFERENCE FROM SIDE STREETS,
  - AN ENTRY IS CALLED TO SWITCH TO ALL RED
  - THE ENTRY IS CONSIDERED URGENT

- UNLESS AN EMERGENCY LIGHT CHANGE IS TO BE PERFORMED, THE TASK SHOULD CHANGE THE LIGHT IF IT NEEDS TO BE CHANGED
  - THE else PART IN A SELECTIVE WAIT STATEMENT PROVIDES THIS CAPABILITY.

INSTRUCTOR NOTES

- NORMALLY, THE ELSE PART OF THE SELECTIVE WAIT STATEMENT IS EXECUTED. THIS SIMPLY CHANGES THE LIGHT IF NECESSARY.

- WHEN ONE OF THE URGENT ENTRY CALLS Start_Emergency_Light_Pattern_1 OR Start_Emergency_Light_Pattern_2 IS ISSUED, THE CORRESPONDING ACCEPT ALTERNATIVE IS SELECTED INSTEAD. THE TRAFFIC LIGHT IS PLACED IN THE EMERGENCY LIGHT PATTERN.

- IN EITHER CASE THE LOOP IS DELAYED UNTIL THE NEXT POLL TIME.

SELECTIVE WAIT WITH AN ELSE PART

- THE else PART IS EXECUTED WHEN NONE OF THE OTHER ALTERNATIVES CAN BE ACCEPTED IMMEDIATELY.

- else PARTS, delay ALTERNATIVES AND terminate ALTERNATIVES ARE MUTUALLY EXCLUSIVE.

```
loop

   select

      accept Start_Emergency_Light_Pattern_1 do
         Set_Emergency_Light_Pattern_1;
      end Start_Emergency_Light_Pattern_1;

   or

      accept Start_Emergency_Light_Pattern_2 do
         Set_Emergency_Light_Pattern_2;
      end Start_Emergency_Light_Pattern_2;

   else

      Monitor_Traffic_Flow;
      Change_Light_If_Necessary;

   end select;

   delay Next_Poll_Time - Calendar.Clock;
   Next_Poll_Time := Next_Poll_Time + 0.5;

end loop;
```

INSTRUCTOR NOTES

- THIS SLIDE PROVIDES MOTIVATION FOR TIMED ENTRY CALLS.

PROBLEM - WHEN AN ENTRY CALL IS NOT ACCEPTED IN TIME

- IF A TASK'S ENTRY IS NOT ACCEPTED WITHIN SOME SPECIFIED TIME, THEN THE TASK MAY WISH TO PERFORM SOME OTHER ACTION.

- SUPPOSE THE TEMPERATURE OF A WATER-COOLED DEVICE IS TO BE MAINTAINED BELOW A CERTAIN TEMPERATURE. IF THE TEMPERATURE RISES ABOVE THAT LEVEL THEN THE WATER FLOW IS TO BE INCREASED UNTIL THE TEMPERATURE FALLS BY 10 DEGREES.

- WHAT HAPPENS IF THE SENSORS THAT MONITOR THE TEMPERATURE BREAK, AND PREVENT THE Temperature_Task FROM RESPONDING?

- WHAT WE NEED IS A WAY TO TAKE SOME CORRECTIVE ACTION IF THE Read ENTRY OF THE Temperature_Task IS NOT ACCEPTED WITHIN SOME REASONABLE TIME.
  - TIMED ENTRY CALLS PROVIDE THIS CAPABILITY.

INSTRUCTOR NOTES

- NOTE THE NESTING OF A TIMED ENTRY CALL WITHIN A TIMED ENTRY CALL.

- TO SIMPLIFY THE EXAMPLE, WE ASSUME THAT WE HAVE NO PROBLEMS WITH THE VALUE.

- EXECUTION OF THE TIMED ENTRY CALL PROCEEDS AS FOLLOWS:
  - THE ACTUAL PARAMETERS, IF ANY, ARE EVALUATED.
  - THE DELAY EXPRESSION IS EVALUATED AND THEN THE ENTRY CALL IS ISSUED.
    - IF THE RENDEZVOUS CAN BE STARTED WITHIN THE SPECIFIED DELAY, THEN IT IS PERFORMED AND THE FIRST SEQUENCE OF STATEMENTS, IF ANY, IS EXECUTED.
    - OTHERWISE, THE ENTRY CALL IS <u>CANCELLED</u> AFTER THE SPECIFIED DELAY HAS EXPIRED, AFTER WHICH THE SECOND SEQUENCE OF STATEMENTS IS EXECUTED.

- MAKE SURE THE CLASS UNDERSTANDS THAT WHILE A SELECTIVE WAIT CAN CONTAIN MORE THAN ONE ACCEPT STATEMENT, A TIMED ENTRY CALL IS FOR ONE ENTRY CALL.

TIMED ENTRY CALLS

- LOOK LIKE A SELECTIVE WAIT, BUT:
  - THERE ARE ONLY TWO "ALTERNATIVES"
  - THE FIRST ALTERNATIVE STARTS WITH AN ENTRY CALL, NOT AN accept STATEMENT
  - THE SECOND "ALTERNATIVE" ALWAYS STARTS WITH A delay STATEMENT.

```
loop
  select
    Temperature_Task.Read (Temperature => Current_Temperature);
    if Current_Temperature >= Maximum_Temperature then

      Water_Control_Task.Increase_Water_Flow;
      while Current_Temperature >= Maximum Temperature - 10.0 loop
        select
          Temperature_Task.Read (Temperature => Current_Temperature);
        or
          delay 0.1;
          raise Temperature_Exception;
        end select;
      end loop;
      Water_Control_Task.Normal_Water_Flow;
    end if;
  or
    delay 0.1;
    raise Temperature_Exception;
  end select;
end loop;
```

- THE SECOND "ALTERNATIVE" IS CHOSEN IF THE ENTRY CALL IS NOT ACCEPTED IN THE SPECIFIED AMOUNT OF TIME, AND THE ENTRY CALL IS CANCELLED.

INSTRUCTOR NOTES

- THIS SLIDE PROVIDES MOTIVATION FOR CONDITIONAL ENTRY CALLS.

PROBLEM - WHEN AN ENTRY CALL CANNOT BE ACCEPTED IMMEDIATELY

- IF A TASK'S ENTRY CALL CANNOT BE ACCEPTED <u>IMMEDIATELY</u>, THEN THE TASK MAY WISH TO PERFORM SOME OTHER ACTIONS:
- SUPPOSE A TASK WANTS TO ALLOCATE SPACE ON A FAST DISK?
- IF THE FASTER DISK IS NOT READY TO ACCEPT ALLOCATION REQUESTS THE CALLING TASK WAITS.
- SUPPOSE THERE IS A SLOWER DISK AVAILABLE THAT THE CALLING TASK WOULD BE WILLING TO USE IT IF THE FASTER ONE IS NOT IMMEDIATELY AVAILABLE.
- WE WANT A WAY TO ASK FOR THE SLOWER DISK IF THE FASTER ONE IS NOT IMMEDIATELY AVAILABLE. IF NEITHER DISK IS AVAILABLE IMMEDIATELY, THEN WE WANT TO PERFORM SOME OTHER ACTION, SUCH AS DELAYING FOR A SECOND BEFORE REQUESTING THE ALLOCATION AGAIN.
  - CONDITIONAL ENTRY CALLS PROVIDE THIS CAPABILITY.

- Fast_Disk IS THE DISK MANAGER TASK FOR A FAST DISK. A CALL ON ITS Allocate_Block ENTRY FINDS A BLOCK OF DISK SPACE, WAITING IF NECESSARY UNTIL ONE BECOMES AVAILABLE. THE NUMBER OF THE ALLOCATED BLOCK IS RETURNED IN Address_Allocated.

- IF SPACE ON THE FASTER DISK IS NOT AVAILABLE IMMEDIATELY, THEN THE TASK SHOWN ON THE SLIDE ATTEMPTS TO OBTAIN SPACE ON THE SLOWER DISK. IF THE SLOWER DISK IS NOT AVAILABLE, THEN WE WAIT FOR A SECOND AND TRY ALL OVER AGAIN.

- TO AVOID BUSY WAITING, WE DELAY FOR ONE SECOND. THIS FREES THE PROCESSOR TO POSSIBLY DO SOME OTHER USEFUL WORK.

- EXECUTION OF A CONDITIONAL ENTRY CALL PROCEEDS AS FOLLOWS:
  - THE ACTUAL PARAMETERS, IF ANY, ARE EVALUATED.
  - THE ENTRY CALL IS ISSUED.
    - IF THE CALLED TASK CAN ESTABLISH A RENDEZVOUS WITH THE CALLING TASK IMMEDIATELY THEN THE RENDEZVOUS TAKES PLACE, AND THE FIRST SEQUENCE OF STATEMENTS IS EXECUTED.
    - OTHERWISE, IF THE RENDEZVOUS CANNOT TAKE PLACE IMMEDIATELY THEN THE ENTRY CALL IS <u>CANCELLED</u> AND THE SECOND SEQUENCE OF STATEMENTS IS EXECUTED.

- THE RENDEZVOUS MAY FAIL TO OCCUR IMMEDIATELY BECAUSE
  - THERE ARE OTHER CALLS QUEUED FOR THE ENTRY, OR
  - THE CALLED TASK IS NOT WAITING AT AN ACCEPT STATEMENT FOR THE ENTRY, AND
  - THE CALLED TASK IS NOT WAITING AT A SELECTIVE WAIT STATEMENT HAVING AN OPEN ACCEPT ALTERNATIVE FOR THE ENTRY.

- AGAIN, MAKE SURE THAT THE STUDENTS UNDERSTAND THAT A CONDITIONAL ENTRY CALL IS FOR ONE ENTRY CALL.

CONDITIONAL ENTRY CALLS

- LOOKS LIKE A SELECTIVE WAIT, BUT:
  - THERE IS ONE ALTERNATIVE PLUS AN else PART
  - THE ALTERNATIVE BEGINS WITH AN ENTRY call RATHER THAN AN accept STATEMENT.

- IF THE CALLED TASK IS NOT ALREADY WAITING TO ACCEPT A CALL TO THE SPECIFIED ENTRY, THE else PART IS EXECUTED AND THE ENTRY CALL CANCELLED.

```
Allocated := False;
loop
  select
     Fast_Disk.Allocate_Block (Address_Allocated => Start_Address);
     Allocated := True;
  else
    select
       Slow_Disk.Allocate_Block Address_Allocated => Start_Address);
       Allocated := True;
    else
       null;
    end select;
  end select;
  exit when Allocated;
  delay 1.0;
end loop;
```

INSTRUCTOR NOTES

- THIS SLIDE JUST SUMS UP THE FORMS OF THE SELECT STATEMENTS FOR
  - ACCEPT STATEMENTS
  - ENTRY CALLS

- POINT OUT AGAIN THAT IN THE SELECTIVE WAIT
  - DELAY ALTERNATIVES
  - TERMINATE ALTERNATIVES
  - ELSE PARTS

  ARE MUTUALLY EXCLUSIVE.

PUTTING IT ALL TOGETHER

- SELECTIVE WAIT STATEMENT (accept STATEMENTS)

```
        select
           [when condition =>]
              selective wait alternative
       { or
           [when condition =>]
              selective wait alternative}
       [else
              sequence of statements]
        end select;
```

WHERE A selective wait alternative HAS ONE OF THE FOLLOWING THREE FORMS:

```
accept statement             |   delay expression;             |   terminate
sequence of statements       |     sequence of statements      |
```

- TIMED ENTRY CALL

```
select
   entry call
   sequence of statements
or
   delay [expression]
      sequence of statements
end select;
```

- CONDITIONAL ENTRY CALL

```
select
   entry call
   sequence of statements
else
   sequence of statements
end select;
```

INSTRUCTOR NOTES

THIS SLIDE BEGINS THE THIRD LOGICAL SUBSECTION, :SURVEY OF OTHER TASKING FEATURES."

THIS IS A VERY HIGH-LEVEL OVERVIEW, WITH ONLY ONE OR TWO SLIDES ON EACH OF THE TOPICS LISTED. IT IS NOT INTENDED TO IMPART COMPLETE UNDERSTANDING OF THESE TOPICS, BUT TO LET STUDENTS KNOW WHAT THEY HAVE YET TO LEARN IF THEY WANT TO ATTAIN A DEEPER MASTERY OF TASKING.

OTHER TASKING FEATURES

- ABORTING TASKS
- EXCEPTIONS IN MULTITASK PROGRAMS
- INTERRUPT-HANDLING TASKS
- "ARRAYS" OF ENTRIES (ENTRY FAMILIES)
- TASK PRIORITIES

INSTRUCTOR NOTES

- BULLET 3:

  TASK NAMES CAN BE COMPLEX NAMES LIKE Task_List (3) OR Task_Pointer.all, FOR EXAMPLE.

- BULLET 4:

  THE abort STATEMENT IS A MEASURE OF LAST RESORT. EVEN IN EMERGENCY SITUATIONS, THE abort STATEMENT MIGHT NOT PROVIDE A WORKABLE SOLUTION.

  - ITEM 3: A TASK ABORTED IN THE MIDDLE OF INSERTING AN ITEM IN A LIST, FOR EXAMPLE, MIGHT LEAVE THE LENGTH OF THE LIST HOLDING AN INCORRECT VALUE.

  - ITEM 4: IN FACT, A TASK STUCK IN A LOOP THAT NEVER PERFORMS ANY OF THESE ACTIONS MIGHT NEVER BE COMPLETE. IT DEPENDS ON THE RUNTIME SYSTEM. ACTIONS THAT FORCE AN ABORTED TASK TO REALLY STOP INCLUDE WAITING FOR OR COMPLETING A RENDEZVOUS, STARTING OR COMPLETING ELABORATION OF THE TASK BODY'S DECLARATION OR TRYING TO ACTIVATE OR ABORT ANOTHER TASK. (A CALLING TASK ABORTED IN THE MIDDLE OF A RENDEZVOUS ALWAYS STAYS ALIVE UNTIL THE RENDEZVOUS ENDS. THIS IS STATED ON THE NEXT SLIDE.)

- BULLET 5:

  - ITEM 3: THIS ASSUMES THAT ANY POSSIBLE RIPPLE EFFECT HAS BEEN ACCOUNTED FOR. THE CALLING TASK SHOULD DELAY BETWEEN THE TIME THE "PLEASE TERMINATE" ENTRY IS CALLED AND THE TIME THE abort STATEMENT IS EXECUTED, TO GIVE THE CALLED TASK A CHANCE TO ACT ON THE ENTRY CALL. THE abort STATEMENT HAS NO EFFECT IF THE TASKS IT NAMES HAVE ALREADY BEEN COMPLETED.

ABORTING TASKS

- THE abort STATEMENT IS AN "EMERGENCY BRAKE"
- IT IS USED TO PREVENT AN OUT-OF-CONTROL TASK FROM INTERFERING WITH THE REST OF A PROGRAM.
- FORM:

  abort $\boxed{\texttt{task name}}$ {, $\boxed{\texttt{task name}}$ };

- THIS STATEMENT IS INTENDED FOR USE ONLY IN EXTREME SITUATIONS, NOT FOR ROUTINE TASK MANAGEMENT.
  - MOST TASKS PLAY AN INDISPENSABLE ROLE IN A PROGRAM
  - ABORTING ONE TASK CAN CREATE A RIPPLE EFFECT, CAUSING TASKS COMMUNICATING WITH THE ABORTED TASK TO FAIL.
  - AN ABORTED TASK HAS NO CHANCE TO EXECUTE ITS "LAST WISHES"
    - LEAVE DATA STRUCTURES IN A CONSISTENT STATE
    - CLEAN UP: DEALLOCATE STORAGE, WRITE AN AUDIT TRAIL, CLOSE FILES, ETC.
  - TAKES EFFECT AT AN IMPRECISELY DEFINED TIME
    - RUNTIME SYSTEM CAN ACTUALLY KEEP AN "ABORTED" TASK ALIVE UNTIL IT TRIES TO PERFORM CERTAIN ACTIONS INVOLVING OTHER TASKS.
  - DUE TO INCREASED COMPLEXITY OF TASK INTERACTION, PROGRAMS WITH abort STATEMENTS ARE HARD TO UNDERSTAND AND VALIDATE.
- A MORE CONTROLLED ALTERNATIVE, APPROPRIATE FOR ROUTINE SITUATIONS, IS A "PLEASE TERMINATE" ENTRY
  - A TASK'S "PLEASE TERMINATE" ENTRY IS CALLED TO ASK IT TO TERMINATE.
  - UPON ACCEPTING THE ENTRY, THE TASK EXECUTES ITS LAST WISHES, <u>THEN</u> TERMINATES.
  - AN abort STATEMENT MIGHT BE PLACED AFTER THE ENTRY CALL AS A BACKUP, IN CASE THE CALLED TASK FAILS TO RESPOND TO THE REQUEST.

INSTRUCTOR NOTES

THE IMPORTANT POINT OF THIS SLIDE IS NOT ITS SUBSTANCE, BUT ITS LENGTH AND COMPLEXITY. GO OVER IT QUICKLY AND AVOID GETTING STUCK ON SPECIFICS.

THE MESSAGE IS THAT MULTITASKING GREATLY COMPLICATES THE RULES FOR EXCEPTIONS.

POINT OUT THE CONTRAST BETWEEN BULLETS 2 AND 3 (EXCEPTIONS RAISED IN TASK BODY DECLARATIVE PART AND TASK BODY STATEMENT SEQUENCE).

- BULLET 4: THE EXCEPTION IS RERAISED IN TWO PLACES. EACH RAISING IS HANDLED OR PROPAGATED SEPARATELY.
- BULLET 6: THE ASYMMETRY CORRESPONDS TO THE ASYMMETRY BETWEEN CALLING AND CALLED TASKS THAT WILL BE DISCUSSED AT THE START OF SECTION 3. THE RULE GIVEN HERE ALLOWS A CALLED TASK TO REMAIN ALIVE TO SERVE OTHER CALLERS.
- BULLET 7: PROGRAMMERS SHOULD TRY TO PREVENT THIS FROM EVER HAPPENING.
- BULLET 8: THIS IS JUST A CONSEQUENCE OF THE RULES FOR TERMINATION THAT WE HAVE ALREADY SEEN.

SPECIAL RULES FOR EXCEPTIONS IN MULTITASK PROGRAMS

- THE PREDEFINED EXCEPTION Tasking_Error IS RAISED IN CERTAIN SITUATIONS INVOLVING TASK INTERACTION.

- IF AN EXCEPTION IS RAISED WHILE ELABORATING THE DECLARATIVE PART OF A TASK BODY:
  - FURTHER PROCESSING OF THE TASK BODY IS ABANDONED.
  - Tasking_Error IS RAISED IN THE PROGRAM UNIT THAT DECLARED OR ALLOCATED THE TASK.

- IF AN EXCEPTION IS RAISED IN THE SEQUENCE OF STATEMENTS OF A TASK BODY (OUTSIDE OF AN accept STATEMENT):
  - THE SEQUENCE OF STATEMENTS IS ABANDONED.
  - IF AN EXCEPTION HANDLER IS PRESENT, IT IS INVOKED.
  - IF NO HANDLER IS PRESENT, THE EXCEPTION IS <u>NOT</u> PROPAGATED.

- IF AN EXCEPTION IS RAISED INSIDE AN accept STATEMENT (I.E. DURING A RENDEZVOUS):
  - FURTHER EXECUTION OF THE accept STATEMENT IS ABANDONED.
  - THE EXCEPTION IS PROPAGATED TO THE ENTRY CALL STATEMENT THAT LED TO THE RENDEZVOUS.
  - THE EXCEPTION IS ALSO RAISED IN THE CALLED TASK, JUST AFTER THE accept STATEMENT.

- IF A TASK ATTEMPTS TO CALL A COMPLETED OR ABORTED TASK (OR IF THE TASK COMPLETES OR IS ABORTED BEFORE IT ACCEPTS THE CALL):
  - Tasking_Error IS RAISED BY THE ENTRY CALL

- IF A TASK IS ABORTED WHILE COMMUNICATING WITH ANOTHER TASK:
  - IF THE CALLING TASK IS ABORTED, THE ENTRY CALL IS CANCELLED OR THE RENDEZVOUS COMPLETES NORMALLY.
  - IF THE CALLED TASK IS ABORTED, Tasking_Error IS RAISED BY THE ENTRY CALL.

- Program_Error IS RAISED BY A select STATEMENT IF IT HAS NO else PART AND EVERY ALTERNATIVE HAS A FALSE GUARD.

- IF AN EXCEPTION IS RAISED IN A SUBPROGRAM, BLOCK STATEMENT, OR TASK BODY THAT IS THE MASTER OF SOME TASK, THE MASTER DOES NOT PROPAGATE THE EXCEPTION UNTIL THE TASK TERMINATES.

- BULLET 3:

  ESSENTIALLY, THE OUTSIDE WORLD IS VIEWED AS A COLLECTION OF TASKS THAT CALL ENTRIES WHEN IT IS TIME TO CAUSE AN INTERRUPT.

- BULLET 4:

  CAVEAT: THIS IS ONLY ONE POSSIBLE IMPLEMENTATION. WE DEPART FROM OUR USUAL POLICY OF NOT DESCRIBING POSSIBLE IMPLEMENTATIONS TO PERSUADE THE STUDENT THAT THIS MECHANISM IS CAPABLE OF HANDLING INTERRUPTS PROMPTLY.

  AN INTERRUPT VECTOR IS A LIST OF ADDRESSES OF THE MACHINE LANGUAGE ROUTINES THAT ARE TO BE GIVEN CONTROL BY THE HARDWARE WHEN INTERRUPTS OCCUR. THE MACHINE ARCHITECTURE ASSIGNS DIFFERENT OFFSETS WITHIN THE VECTOR TO HOLD THE ADDRESSES OF THE ROUTINES FOR DIFFERENT KINDS OF INTERRUPTS.

HANDLING HARDWARE INTERRUPTS

- HARDWARE INTERRUPTS CAN BE HANDLED BY Ada TASKS.

- AN ENTRY OF A TASK CAN BE DESIGNATED TO BE AN <u>INTERRUPT ENTRY</u>.
  - WHEN THE SPECIFIED INTERRUPT OCCURS, THE DESIGNATED INTERRUPT IS CALLED.
  - THE INTERRUPT IS HANDLED BY ACCEPTING A CALL ON THAT ENTRY.

- THIS PROVIDES A HIGH-LEVEL VIEW OF INTERRUPTS.
  - CONSISTENT WITH USUAL MEANS OF TASK SYNCHRONIZATION AND COMMUNICATION.
  - ALLOWS DEVICE DRIVERS TO BE WRITTEN AT A HIGH LEVEL OF ABSTRACTION.

- HANDLING INTERRUPTS IN THIS WAY CAN BE QUITE EFFICIENT.
  - ONE POSSIBLE IMPLEMENTATION:
    - WHEN A TASK IS READY TO ACCEPT A "CALL" ON THE INTERRUPT ENTRY, IT STORES THE ADDRESS OF THE accept STATEMENT'S OBJECT CODE IN THE HARDWARE INTERRUPT VECTOR FOR THE DESIGNATED INTERRUPT.
    - UPON AN INTERRUPT, THE HARDWARE BRANCHES DIRECTLY TO THE accept STATEMENT.

INSTRUCTOR NOTES

THE LINE BEGINNING "for Serial_Interface_Interrupt" IS AN ADDRESS CLAUSE. IT IS A LOW-LEVEL FEATURE THAT CAN ALSO BE USED TO SPECIFY ADDRESSES OF DATA OBJECTS OR CODE FOR PROGRAM UNITS.

THE NUMERIC LITERAL 16#40# IS A BASED NUMBER. IT MEANS 40 HEXADECIMAL (EQUAL TO 64 DECIMAL).

DESIGNATING INTERRUPT ENTRIES

- AN ENTRY MAY BE DESIGNATED AS AN INTERRUPT ENTRY BY SPECIFYING ITS "ADDRESS"
  - THE ADDRESS REFERS TO A PARTICULAR INTERRUPT, ACCORDING TO IMPLEMENTATION-DEFINED CONVENTIONS (TYPICALLY THE ADDRESS OF THE CORRESPONDING HARDWARE INTERRUPT VECTOR).
  - THE SPECIFICATION OF THE ADDRESS APPEARS IN THE TASK DECLARATION, SOMEPLACE AFTER THE ENTRY DECLARATION.
  - THE TASK DECLARATION MUST BE IN THE SCOPE OF A with CLAUSE FOR THE PACKAGE System.

- EXAMPLE:

```
with System;

package Serial_Interface_Package is

   task Serial_Interface_Task is
      entry Send_Character (C : in Character);  -- ordinary entry
      entry Serial_Interface_Interrupt;          -- interrupt entry
      for Serial_Interface_Interrupt use at 16#40#;
   end Serial_Interface_Task;

   Serial_Interface_Malfunction : exception;

end Serial_Interface_Package;
```

- Serial_Interface_Interrupt IS ACCEPTED LIKE ANY OTHER ENTRY.

INSTRUCTOR NOTES

- BULLET 1:

  WE ARE USING THE WORD ARRAY LOOSELY HERE. THE Ada REFERENCE MANUAL RESERVES THE TERM TO MEAN AN ARRAY OF OBJECTS IN SOME DATA TYPE.

  DISTINGUISH BETWEEN AN ARRAY OF TASK OBJECTS AND A TASK OBJECT WITH AN ARRAY OF ENTRIES. (THE FORMER HAS SEVERAL THREADS OF CONTROL, THE LATTER ONLY ONE.)

- BULLET 2:

  THE TASK HAS SEVEN ENTRIES, NAMED Request, Reserve (1), Reserve (2), Reserve (3), Reserve (4), Reserve (5), Release.

- BULLET 3:

  THE FAMILY MEMBER TO BE CALLED CAN BE INDEXED BY AN EXPRESSION COMPUTED AT RUNTIME.

- BULLET 4:

  EACH FAMILY MEMBER HAS ITS OWN QUEUE OF ENTRY CALLS. THESE LOOPS CHECK FOR ALL CALLS IN THE QUEUE OF Reserve (5), ALL CALLS IN THE QUEUE OF Reserve (4), ETC., BUT STOP CHECKING FOR CALLS ON Reserve (n) ONCE Available_Count HAS FALLEN BELOW n.

  THIS IS PART OF A RESOURCE MANAGER. ENTRY FAMILIES CAN ALSO BE USED FOR MAINTAINING QUEUES OF ENTRY CALLS WITH DIFFERENT LEVELS OF PRECEDENCE. THUS ENTRY FAMILIES ALLOW GREATER FLEXIBILITY IN THE SCHEDULING OF RENDEZVOUS.

ENTRY FAMILIES

- AN ENTRY FAMILY ACTS LIKE A ONE-DIMENSIONAL "ARRAY OF ENTRIES."
  - INDIVIDUAL ENTRIES OF A FAMILY ARE IDENTIFIED BY INDEX VALUES.
  - ALLOWS SEVERAL ENTRIES OF THE SAME TASK OBJECT TO BE TREATED UNIFORMLY.

- DECLARATION:

```
task Frequency_Monitor is
   entry Request (Request_Size : in Frequency_Count_Subtype);
   entry Reserve (1 .. 5) (Frequency_List : out Frequency_List_Type);
   entry Release (Frequency_List : in Frequency_List_Type);
end Frequency_Monitor;
```

- ENTRY CALL:

```
procedure Reserve_Frequencies (Frequency_List : out Frequency_List_Type) is
begin
   Frequency_Monitor.Request (Frequency_List'Length);
   Frequency_Monitor.Reserve (Frequency_List'Length) (Frequency_List);
end Reserve_Frequencies;
```

- accept STATEMENT:

```
for Next_Try in reverse 1 .. Available_Count loop
   while Available_Count >= Next_Try loop
      select
         accept Reserve (Next_Try) (Frequency_List : out Frequency_List_Type) do
            Frequency_List :=
               Available_List : (Available_Count-Request_Size+1 .. Available_Count);
            Available_Count := Available_Count - Request_Size;
         end Reserve;
      else
         exit;
      end select;
   end loop;
end loop;
```

INSTRUCTOR NOTES

- BULLET 1:

  THE PRIORITY IS FIXED AT COMPILE-TIME. PRIORITIES MAY BE ASSIGNED TO SINGLE TASKS OR TO TASK TYPES. ALL TASKS OF THE SAME TYPE HAVE THE SAME PRIORITY.

- BULLET 2:

  - ITEMS 2 AND 3 ARE SIMPLY CONSEQUENCES OF ITEM 1.
  - ITEM 4:

    THIS IS BECAUSE PROGRESS OF BOTH TASKS IS AFFECTED BY HOW FAST THE RENDEZVOUS GETS EXECUTED.
  - ITEM 5:

    INTERRUPTS ARE ALWAYS HANDLED IMMEDIATELY UNLESS ANOTHER INTERRUPT IS ALREADY BEING HANDLED OR THE INTERRUPT HANDLING TASK IS NOT READY.

- BULLET 3:

  THE EFFECT OF PRIORITIES IS VERY NARROWLY DEFINED. THEY SHOULD BE USED TO INDICATE RELATIVE DEGREES OF URGENCY, NOT TO SYNCHRONIZE TASKS.

TASK PRIORITIES

- A TASK MAY BE ASSIGNED A PERMANENT PRIORITY INDICATING HOW URGENT ITS EXECUTION IS:

```
task type Temperature_Sensor_Task_Type is
   pragma Priority (10);
   entry Get_Temperature (Temperature : out Temperature_Type);
end Temperature_Sensor_Task_Type;
```

  - HIGHER NUMBER = GREATER URGENCY

- WHAT PRIORITIES DO:

  - WHEN TWO OR MORE TASKS ARE READY TO EXECUTE AND THE RUNTIME SYSTEM MUST DECIDE WHICH TASK WILL GET THE PROCESSOR NEXT, THE TASK WITH HIGHEST PRIORITY WINS.
  - IN A SINGLE-PROCESSOR SYSTEM, A LOWER PRIORITY TASK IS PREEMPTED WHEN A HIGHER PRIORITY TASK STOPS WAITING FOR A RENDEZVOUS, THE END OF A DELAY, OR THE TERMINATION OF ANOTHER TASK.
  - IN A MULTIPROCESSOR SYSTEM, TASKS OF LOWER AND HIGHER PRIORITIES MAY BE RUNNING SIMULTANEOUSLY.
  - A RENDEZVOUS IS EXECUTED AT THE PRIORITY OF ITS HIGHER-PRIORITY PARTICIPANT.
  - RENDEZVOUS WITH INTERRUPT ENTRIES ARE EXECUTED AT A HIGHER PRIORITY THAN ANYTHING ELSE.

- WHAT PRIORITIES DON'T DO:

  - THEY DON'T AFFECT WHICH CALL ON AN ENTRY IS ACCEPTED FIRST. (ENTRY CALLS ARE ALWAYS QUEUED FIRST-COME-FIRST-SERVED.)
  - THEY DON'T AFFECT WHICH ALTERNATIVE OF A SELECTIVE WAIT IS CHOSEN.
  - ON A MULTIPROCESSOR SYSTEM, EXECUTION OF A HIGHER PRIORITY TASK DOES NOT BLOCK EXECUTION OF A LOWER PRIORITY TASK.

INSTRUCTOR NOTES

ALLOW 90 MINUTES FOR THIS SECTION -- 60 MINUTES BEFORE THE BREAK AND 30 MINUTES AFTER.

THE BREAK SHOULD OCCUR JUST BEFORE THE CYCLIC PROCESSING TOPIC.

SECTION 3

FUNDAMENTAL TASK DESIGNS

VG 831

INSTRUCTOR NOTES

NOW THAT STUDENTS HAVE SEEN THE BUILDING BLOCKS FOR MULTITASK Ada PROGRAMS, THIS SECTION WILL DISCUSS WAYS OF COMBINING THOSE BUILDING BLOCKS.

WE WILL SPEND ABOUT 15 MINUTES ON EACH OF THE FIRST FOUR TOPICS AND ABOUT 30 MINUTES ON THE LAST.

THERE WILL BE A BREAK AFTER THE FIRST FOUR TOPICS.

- BULLET 1: THIS TOPIC IS CONCERNED WITH THE DIFFERENT ROLES PLAYED BY CALLING TASKS AND ACCEPTING TASKS.
- BULLET 2: MONITORS ARE A MECHANISM FOR PREVENTING SIMULTANEOUS UPDATE OF DATA SHARED BY TWO OR MORE TASKS.
- BULLET 3: MESSAGE BUFFERS ARE A MORE FLEXIBLE MEANS OF INTERTASK COMMUNICATION THAT CAN BE IMPLEMENTED IN TERMS OF RENDEZVOUS.
- BULLET 4: STREAM-ORIENTED TASK DESIGN SOLVES CERTAIN PROGRAMMING PROBLEMS BY HAVING TASKS CONSUME STREAMS OF INPUT DATA PRODUCED BY OTHER TASKS AND PRODUCE STREAMS OF OUTPUT DATA TO BE CONSUMED BY OTHER TASKS.
- BULLET 5: TRADITIONALLY, CYCLIC PROCESSING IS DONE BY A CYCLIC EXECUTIVE, BUT Ada ALLOWS AN ALTERNATIVE APPROACH.

DESIGNS TO BE CONSIDERED

- SERVER AND USER TASKS
- MONITORS
- MESSAGE BUFFERS
- STREAM-ORIENTED TASK DESIGN
- CYCLIC PROCESSING

INSTRUCTOR NOTES

THIS SLIDE GIVES THE MAIN MESSAGE ABOUT SERVER AND USER TASKS.

- BULLET 2: THE EXAMPLE WAS FIRST INTRODUCED IN SECTION 2.
- BULLET 3: THE NEXT SLIDE PURSUES THE ANALOGY BETWEEN PROCEDURES AND ENTRIES.

SERVER AND USER TASKS

- WHEN TASK A CALLS AN ENTRY OF TASK B, WE CAN USUALLY VIEW TASK B AS PROVIDING SOME SERVICE TO TASK A.
  - TASK A REQUESTS THE SERVICE BY CALLING THE ENTRY.
  - TASK B PROVIDES THE SERVICE BY EXECUTING AN ACCEPT STATEMENT FOR THE ENTRY.
- A TASK'S ENTRIES CORRESPOND TO THE SERVICES THAT A TASK PROVIDES.

```
task type Shared_Count_Type is
   entry Increase_Count (By : in Positive);
   entry Get_Count (Sum_So_Far : out Natural);
end Shared_Count_Type;
```

- FROM THE CALLING TASK'S POINT OF VIEW, AN ENTRY CALL IS VERY MUCH LIKE A PROCEDURE CALL.
  - CONTEXT:

```
Event_Count : Shared_Count_Type;
N           : Integer;
```

  - EXAMPLES:

```
Event_Count.Increase_Count (4);
Event_Count.Get_Count (N);
```

  - EXECUTION OF A CALL STATEMENT CAUSES SOME SERVICE TO BE PERFORMED.
  - THE SERVICE TO BE PERFORMED MAY BE SPECIFIED BY <u>in</u> OR <u>in out</u> PARAMETERS.
  - PERFORMANCE OF THE SERVICE MAY BE REFLECTED IN THE SETTING OF <u>in out</u> OR <u>out</u> PARAMETERS.

INSTRUCTOR NOTES

- BULLET 1: THE CONCEPTUAL SIMILARITY WAS EXPLAINED ON THE PREVIOUS SLIDE.

- BULLET 2:

  - ITEM 3:

    ESSENTIALLY, A PARAMETER-TYPE PROFILE OF A PROCEDURE OR ENTRY IS AN ORDERED LIST OF THE BASE TYPES OF THE PARAMETERS. (FUNCTIONS HAVE PARAMETER/RESULT TYPE PROFILES.)

    THE EXAMPLES ASSUMES THAT A SYSTEM HAS EVOLVED OVER THE YEARS TO INCLUDE BOTH OLD AND NEW MODELS OF A CERTAIN KIND OF SENSOR. THE NEW MODEL PROVIDES MORE PRECISE DATA. THERE ARE FIXED-POINT TYPES DECLARED FOR THE DATA PROVIDED BY EACH MODEL.

    EACH SENSOR HAS A TASK THAT CALLS Data_Analysis_Task.Report_Sensor_Reading WITH EACH PIECE OF DATA IT OBTAINS. THERE ARE ACTUALLY TWO SEPARATE ENTRIES BY THAT NAME, AND THE TYPE OF THE ACTUAL PARAMETER DETERMINES WHICH ONE IS CALLED. SIMILARLY, Data_Analysis_Task SHOULD HAVE AT LEAST TWO ACCEPT STATEMENTS FOR Report_Sensor_Reading, ONE WITH EACH FORMAL PARAMETER TYPE. THE TYPE OF THE FORMAL PARAMETER DETERMINES WHICH Report_Sensor_Reading ENTRY THE ACCEPT STATEMENT APPLIES TO.

    OVERLOADING ALLOWS TASKS FOR BOTH KINDS OF SENSORS TO BE WRITTEN IN THE SAME WAY.

  - ITEM 4:

    AN ENTRY WITH MATCHING PARAMETER TYPES AND MODES MAY BE USED AS THE "OLD NAME" IN A RENAMING DECLARATION FOR A PROCEDURE. AN ENTRY MAY BE USED AS A GENERIC ACTUAL PARAMETER CORRESPONDING TO A GENERIC FORMAL PROCEDURE WITH MATCHING PARAMETER TYPES AND MODES.

SIMILARITIES BETWEEN ENTRIES AND PROCEDURES

- CONCEPTUAL SIMILARITY:
  - ENTRY CALLS AND PROCEDURE CALLS ARE BOTH WAYS TO OBTAIN A SERVICE.

- STRUCTURAL SIMILARITIES:
  - SIMILAR CALLING SYNTAX
    - ONLY *DIFFERENCE* IS HOW THE SERVER IS NAMED:

```
[procedure name] [( [actual parameter list] )];

[task name] . [entry name] [( [actual parameter list] )];
```

    - ACTUAL PARAMETER LIST MAY BE NAMED, POSITIONAL, OR MIXED.

  - SAME PARAMETER MECHANISM
    - ACTUAL PARAMETERS IN USER CORRESPOND TO FORMAL PARAMETERS IN SERVER
    - FORMAL PARAMETERS HAVE MODE <u>in</u>, <u>out</u>, OR <u>in out</u>.
    - PARAMETERS WITH MODE <u>in</u> MAY BE GIVEN DEFAULT VALUES:

```
entry Increase_Count (By : in Positive := 1);
-- The call Event_Count.Increase_Count is equivalent to
-- Event_Count.Increase_Count (1);
```

  - SAME OVERLOADING RULES
    - ENTRIES WITH DIFFERENT PARAMETER-TYPE "PROFILES" MAY BE OVERLOADED:

```
type Model_1_Sensor_Reading_Type is delta 0.1 range 0.0 .. 1500.0;
type Model_2_Sensor_Reading_Type is delta 0.01 range 0.0 .. 2.0E8;
task Data_Analysis_Task is
   entry Report_Sensor_Reading (Data : in Model_1_Sensor_Reading_Type);
   entry Report_Sensor_Reading (Data : in Model_2_Sensor_Reading_Type);
   ...
end Data_Analysis_Task;
```

  - ENTRIES MAY BE USED AS PROCEDURES IN RENAMING DECLARATIONS AND GENERIC INSTANTIATIONS.

INSTRUCTOR NOTES

- BULLET 1:

  - ITEM 2:

    AN ACCEPT STATEMENT DOES NOT REFER IN ANY WAY TO THE TASK CALLING THE ENTRY. (OF COURSE ONE OF THE ENTRY'S PARAMETERS COULD CONTAIN INFORMATION IDENTIFYING THE CALLER.)

- BULLET 3:

BECAUSE CALLING TASKS ARE USERS, THEY REQUEST A SPECIFIC SERVICE FROM A SPECIFIC TASK. CALLED TASKS CAN FULFILL REQUESTS FOR SERVICE WITHOUT REGARD TO WHO MADE THE REQUEST.

FURTHERMORE, A USER TASK GENERALLY NEEDS A SPECIFIC SERVICE PERFORMED AT A PARTICULAR POINT IN ITS ALGORITHM BEFORE IT CAN PROCEED, WHILE A SERVER TASK CAN WAIT TO SERVE WHICHEVER REQUEST ARRIVES NEXT.

RENDEZVOUS ARE ASYMMETRIC

- KNOWLEDGE OF THE OTHER PARTY TO THE RENDEZVOUS
  - A CALLING TASK NAMES THE TASK WHOSE ENTRY IT IS CALLING:

```
Event_Count, Response_Count : Shared_Count_Type;
   ...
Event_Count.Increase_Count (N);
Response_Count.Increase_Count (1);
```

  - A CALLED TASK HAS NO MECHANISM FOR DETERMINING WHICH TASK ISSUED THE ENTRY CALL IT IS ACCEPTING.

```
accept Increase_Count (By : in Positive) do
   Sum := Sum + By;
end Increase_Count;
```

- WAITING FOR A RENDEZVOUS
  - A CALLING TASK CAN WAIT TO <u>ACCEPT</u> A CALL ON WHICHEVER OF SEVERAL ENTRIES IS CALLED FIRST:

```
select
   accept Increase_Count (By : in Positive) do
      Sum := Sum + By;
   end Increase_Count;
or
   accept Get_Count (Sum_So_Far : out Natural) do
      Sum_So_Far := Sum;
   end Get_Count;
end select;
```

  - A CALLING TASK CAN ONLY HAVE ONE ENTRY <u>CALL</u> WAITING TO BE ACCEPTED AT ANYTIME.

- REASON FOR DIFFERENCES:
  - CALLED TASKS ARE "SERVERS"
  - CALLING TASKS ARE "USERS"

INSTRUCTOR NOTES

- BULLET 2:

  OFTEN, EITHER TASK COULD JUSTIFIABLY BE SEEN AS SERVING THE OTHER. THE TASK DESIGN DEPENDS ON WHICH VIEWPOINT IS CHOSEN.

- BULLET 3:
  - ITEM 1:

    A USER TASK MUST EXPLICITLY NAME EACH TASK IT CALLS, BUT A SERVER TASK ACCEPTS CALLS FROM WHOEVER ISSUES THEM.
  - ITEM 2:

    A USER TASK ISSUED CALLS IN A PARTICULAR ORDER, BUT A SERVER TASK CAN CONTAIN A SELECTIVE WAIT ACCEPTING CALLS ON DIFFERENT ENTRIES AS THOSE CALLS ARE ISSUED.
  - ITEM 3:

    A SERVER TASK TYPICALLY WAITS IN A SELECT STATEMENT UNTIL SOME REQUEST FOR SERVICE ARRIVES, WHILE A USER TASK REQUIRES SERVICE AT A PARTICULAR POINT IN ITS PROCESSING. THUS THE USER TASK INITIATES THE RENDEZVOUS BY CALLING AN ENTRY.
  - ITEM 4

    AN accept STATEMENT CAN ONLY APPEAR DIRECTLY IN THE SEQUENCE OF STATEMENTS OF A TASK BODY, NOT IN A SUBPROGRAM. AN ENTRY CALL CAN APPEAR IN <u>ANY</u> SEQUENCE OF STATEMENTS. IF THE INTERACTION OCCURS IN SEVERAL PLACES, SURROUNDED BY THE SAME STATEMENTS IN EACH PLACE, THOSE STATEMENTS ARE AN APPROPRIATE CANDIDATE FOR A PROCEDURE BODY.

REVERSING THE DIRECTION OF RENDEZVOUS

DESIGN HINT

SOMETIMES THE DESIGN OF A MULTITASK PROGRAM CAN BE SIMPLIFIED BY REVERSING THE DIRECTION OF RENDEZVOUS BETWEEN TWO TASKS.

- INSTEAD OF TASK A CALLING AN ENTRY OF TASK B, TASK B CALLS AN ENTRY OF TASK A.
- THIS ENTAILS RECONSIDERING WHICH TASK SERVES WHICH WHEN THE TWO TASKS COMMUNICATE.
- ISSUES TO CONSIDER
  - WILL ONE TASK BE INTERACTING IN THE SAME WAY WITH MANY DIFFERENT TASKS? (YES => SERVER)
  - WILL THE TASK BE PERFORMING DIFFERENT KINDS OF INTERACTIONS IN AN UNPREDICTABLE ORDER? (YES => SERVER)
  - DOES THE TASK TAKE THE INITIATIVE IN DETERMINING WHEN AN INTERACTION SHOULD TAKE PLACE? (YES => USER)
  - SHOULD THE TASK PERFORM THE INTERACTION FROM WITHIN A SUBPROGRAM? (YES => USER)

INSTRUCTOR NOTES

- BULLET 1:
  - ITEM 1:
    THE PULSES COULD BE RADAR BLIPS OR GEIGERS, FOR EXAMPLE.
  - ITEM 3:
    ASSUME THAT PULSES ARE QUEUED, SO THAT Wait_For_Pulse (N) DOES NOT WAIT IF A PULSE HAS ALREADY BEEN SENSED BY SENSOR N. SIMILARLY, IF 3 PULSES HAVE ARRIVED, Wait_For_Pulse CAN BE CALLED THREE TIMES WITHOUT WAITING. (BRING THIS UP ONLY IF ASKED.)

- BULLET 2:
  - ITEM 2:
    ONE OF THESE TASKS IS ASSIGNED TO EACH SENSOR, TO INCREMENT *Unreported_Pulses EACH TIME A PULSE IS RECEIVED BY THAT SENSOR.*
  - ITEM 3:
    THE FUNCTION SAVES THE VALUE OF Unreported_Pulses, RESETS THAT VARIABLE TO ZERO (TO START COUNTING THE PULSES THAT WILL BE REPORTED BY THE NEXT FUNCTION CALL) AND RETURNS THE SAVED VALUE.

- BULLET 3:
  THE CLASS SHOULD RECOGNIZE BY NOW THAT THIS IS A SIMULTANEOUS UPDATE PROBLEM. WITH LUCK, THEY MAY BE ABLE TO POINT OUT SOME OF THE SPECIFIC THINGS THAT CAN GO WRONG:
  - IF ONE OF THE SENSOR TASKS INCREMENTS Unreported_Pulses WHILE New_Pulses IS BETWEEN ITS FIRST AND SECOND STATEMENTS, THE PULSE WILL NOT BE COUNTED.
  - IF TWO SENSOR TASKS BOTH EVALUATE THE RIGHTHAND SIDES OF THE ASSIGNMENT STATEMENTS, THEN BOTH PERFORM THE ASSIGNMENT, Unreported_Pulses WILL BE INCREMENTED BY 2 INSTEAD OF 1.
  - IF Unreported_Pulses HAS THE VALUE 99, A SENSOR TASK EVALUATES THE RIGHTHAND SIDE OF THE ASSIGNMENT, New_Pulses RESETS Unreported_Pulses to 0, AND THE SENSOR TASK COMPLETES THE ASSIGNMENT, Unreported_Pulses WILL HAVE THE VALUE 100 INSTEAD OF 1.

MONITORS: THE SIMULTANEOUS UPDATE PROBLEM REVISITED

- PROBLEM:
  - FIVE SENSORS, EACH RECEIVING PULSES
  - NEED A FUNCTION New_Pulses RETURNING THE TOTAL NUMBER OF PULSES RECEIVED ON ALL SENSORS SINCE THE LAST CALL ON THE FUNCTION.
  - INTERFACE: THE PROCEDURE CALL Wait_For_Pulse (N) CAUSES THE TASK CALLING IT TO WAIT UNTIL A PULSE IS DETECTED BY SENSOR N, THEN PROCEED.

- AN INCORRECT SOLUTION:
  - A GLOBAL VARIABLE:

```
Unreported_Pulses : Natural := 0;
```

  - FIVE TASKS, EACH WITH ITS OWN VALUE FOR My_Sensor, EXECUTING THE FOLLOWING LOOP:

```
loop
   Wait_For_Pulse (My_Sensor);
   Unreported_Pulses := Unreported_Pulses + 1;
end loop;
```

  - THE New_Pulses FUNCTION BODY (EXECUTED BY A SIXTH TASK):

```
function New_Pulses return Natural is
   Result : Natural;
begin
   Result := Unreported_Pulses;
   Unreported_Pulses := 0;
   return Result;
end New_Pulses;
```

- WHY WON'T THIS WORK?

- BULLET 1:

  THE TASK Count_Manager HAS TWO ENTRIES, Increment_Count AND Obtain_And_Reset_Count. IT MAINTAINS A LOCAL VARIABLE NAMED Current_Count.

  A CALL ON Increment_Count CAUSES Current_Count TO BE INCREMENTED.
  A CALL ON Obtain_And_Reset_Count DELIVERS THE VALUE OF Current_Count TO THE CALLER. Current_Count IS THEN RESET TO ZERO.
  ALL Count_Manager EVER DOES IS SERVICE CALLS ON THESE ENTRIES.

- BULLET 2:

  THE SENSOR TASK NOW CALLS Count_Manager.Increment_Count INSTEAD OF EXECUTING Unreported_Pulses:=Unreported_Pulses + 1.

- BULLET 3:

  New_Pulses NOW CALLS Count_Manager.Obtain_And_Reset_Count (Result) INSTEAD OF EXECUTING

```
Result:=Unreported_Pulses;
Unreported_Pulses:=0;
```

THIS APPROACH ENFORCES MUTUAL EXCLUSION BECAUSE Count_Manager IS THE ONLY TASK WITH ACCESS TO THE VARIABLE Current_Count. UPON ACCEPTING A CALL ON Increment_Count, Count Manager COMPLETES THE INCREMENT OPERATION IN ITS ENTIRETY BEFORE ACCEPTING ANOTHER ENTRY CALL. UPON ACCEPTING A CALL ON Obtain_And_Reset_Count, Count_Manager COMPLETES THE REPORT-OLD-VALUE-AND-RESET OPERATION IN ITS ENTIRETY BEFORE ACCEPTING ANOTHER ENTRY CALL. SINCE Count_Manager CANNOT BE DOING TWO THINGS AT ONCE, THE OPERATIONS ON Current Count CANNOT BE INTERLEAVED, SO THE PROGRAM WORKS CORRECTLY.

SOLUTION: A NEW TASK TO MANAGE THE SHARED DATA

- ADD THE FOLLOWING TASK:

```
task Count_Manager is
   entry Increment_Count;
   entry Obtain_And_Reset_Count (Old_Count : out Natural);
end Count_Manager;

task body Count_Manager is
   Current_Count : Natural := 0;
begin
   loop
      select
         accept Increment_Count;
         Current_Count := Current_Count + 1;
      or
         accept Obtain_And_Reset_Count (Old_Count : out Natural) do
            Old_Count := Current_Count;
         end Obtain_And_Reset_Count;
         Current_Count := 0;
      end select;
   end loop;
end Count_Manager;
```

- CHANGE THE SENSOR TASK LOOP TO:

```
loop
   Wait_For_Pulse (My_Sensor);
   Count_Manager.Increment_Count;
end loop;
```

- CHANGE New_Pulses TO:

```
function New_Pulses return Natural is
   Result : Natural;
begin
   Count_Manager.Obtain_And_Reset_Count (Result);
   return Result;
end New_Pulses;
```

INSTRUCTOR NOTES

THE TERM <u>MONITOR</u> WAS ORIGINALLY INTRODUCED BY C.A.R. HOARE (<u>COMMUNICATIONS OF THE ACM</u>, OCTOBER 1974, pp. 549-557), WHO USED IT TO DESCRIBE A SPECIFIC, SOMEWHAT MORE INTRICATE MECHANISM FOR PROVIDING RESTRICTED, MUTUALLY EXCLUSIVE OPERATIONS ON SOME DATA STRUCTURE. PER BRINCH-HANSEN (<u>OPERATING SYSTEM PRINCIPLES</u>, PRENTICE-HALL, 1973, p. 336) DEFINES A MONITOR MORE GENERALLY AS "A COMMON DATA STRUCTURE AND A SET OF MEANINGFUL OPERATIONS ON IT THAT EXCLUDE ONE ANOTHER IN TIME AND CONTROL THE SYNCHRONIZATION OF CONCURRENT PROCESSES." THAT IS THE DEFINITION USED IN L303. (PROCESS "SYNCHRONIZATION" SIMPLY REFERS TO THE FACT THAT ONE PROCESS MAY BE FORCED TO WAIT WHILE THE MONITOR SERVICES ANOTHER TASK.)

IN THE CASE OF Count_Manager, THE DATA STRUCTURE IS A SIMPLE VARIABLE OF SUBTYPE Natural.

DEFINITION OF A MONITOR

- THE Count_Manager TASK IS AN EXAMPLE OF A MONITOR.

- GENERAL DEFINITION:

  IN Ada, A MONITOR IS A TASK DESIGNED TO RESTRICT ACCESS TO A DATA STRUCTURE.

  - THE DATA STRUCTURE IS DECLARED IN THE TASK BODY.
  - THE DATA STRUCTURE CAN ONLY BE MANIPULATED THROUGH THE ABSTRACT OPERATIONS IMPLEMENTED AS ENTRIES TO THE MONITOR.
  - THE DATA STRUCTURE CANNOT BE SIMULTANEOUSLY MANIPULATED BY MORE THAN ONE TASK.

INSTRUCTOR NOTES

- BULLET 1:

  IN THE CASE OF MONITORS, THE NOTION OF TASK AS DATA OBJECT IS QUITE NATURAL.

  - ITEM 1: IF THE Count_Manager TASK DECLARATION (FOR A SINGLE OBJECT IN AN ANONYMOUS TASK TYPE) IS MADE INTO A TASK TYPE DECLARATION, WE MIGHT NAME THE TYPE Shared_Accumulator_Type. WE COULD THEN DECLARE Pulse_Count TO BE AN OBJECT IN THIS TYPE, AS SHOWN. (MONITOR OBJECTS ARE SPECIAL KINDS OF DATA OBJECTS THAT CAN BE SAFELY SHARED BY CONCURRENT TASKS.)

  - ITEM 2: ONE OF THE OPERATIONS FOR Shared_Accumulator_Type OBJECTS IS TO CALL AN OBJECT'S Obtain_And_Reset_Count ENTRY.

- BULLET 2:

  RECALL FROM SECTION 2 THAT A PROGRAM UNIT OR BLOCK CONTAINING A DECLARATION OF A TASK OBJECT CANNOT TERMINATE UNTIL THE TASK TERMINATES.

- BULLET 3:

  POINT OUT THE WORD TYPE IN THE TASK TYPE DECLARATION.
  EXPLAIN THAT THERE IS ONE ENTRY FOR EACH ABSTRACT OPERATION.
  POINT OUT THE DECLARATION OF LOCAL DATA.
  EXPLAIN THAT THERE IS ONE ACCEPT ALTERNATIVE FOR EACH ENTRY, TO MANIPULATE THE LOCAL DATA IN ACCORDANCE WITH THE CORRESPONDING ABSTRACT OPERATION. GUARDS ARE USED FOR ABSTRACT OPERATIONS THAT ARE ONLY MEANINGFUL WHEN CERTAIN PRECONDITIONS HOLD. (FOR EXAMPLE, EXAMINING THE FIRST ELEMENT IN A LIST IS ONLY MEANINGFUL WHEN THE LIST IS NOT EMPTY.) IF AN ENTRY IS CALLED WHEN ITS PRECONDITION DOES NOT HOLD, THE CALL WILL NOT BE ACCEPTED. THE CALLER WILL BE FORCED TO WAIT UNTIL ITS PRECONDITION DOES HOLD.
  POINT OUT THE TERMINATE ALTERNATIVE.

GUIDELINES FOR MONITORS

- IT IS OFTEN APPROPRIATE TO DECLARE MONITOR TASK TYPES.
  - WE CAN THINK OF THE OBJECTS IN THIS TYPE AS DATA OBJECTS THEMSELVES RATHER THAN AS PROGRAM UNITS.

    ```
    Pulse_Count : Shared_Accumulator_Type;
    ```
  - OPERATIONS ON THE OBJECT TAKE THE FORM OF CALLING ONE OF ITS ENTRIES:

    ```
    Pulse_Count.Obtain_And_Reset_Count (X);
    ```
- MONITOR TASK BODIES SHOULD HAVE terminate ALTERNATIVES, SO PROGRAM UNITS DECLARING OBJECTS IN MONITOR TYPES WILL BE ABLE TO TERMINATE.
- TYPICAL PATTERN OF A MONITOR:

```
task type identifier is
{ entry declaration for an abstract
  operation on shared data }
end identifier ;
```

```
task body identifier is
   declarations, including data to be
   protected from simultaneous update
begin -- identifier
   loop
      select
       {   [when condition =>]
              accept alternative for one
              of the abstract operations
        or }
           terminate;
      end select;
   end loop;
end identifier ;
```

INSTRUCTOR NOTES

- BULLET 2:

  THIS POINT IS CRUCIAL. MAKE SURE STUDENTS UNDERSTAND IT.

  - ITEM 2: THE LOGIC OF THE SUBPROGRAM BODIES MAY BE BASED ON THE ASSUMPTION THAT ONE SUBPROGRAM COMPLETES BEFORE ANOTHER STARTS.

- BULLET 3:

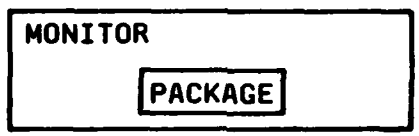
  - ITEM 1: THIS APPROACH INVOLVES PUTTING THE PACKAGE INSIDE THE MONITOR TASK BODY (OR COMPILING IT SEPARATELY AND USING IT THERE). THE MONITOR'S ACCEPT ALTERNATIVES CALL THE PACKAGE'S SUBPROGRAMS, BUT THE MONITOR MAKES SURE THAT ONLY ONE SUBPROGRAM IS CALLED AT A TIME.

  - ITEM 2: THIS APPROACH INVOLVES PUTTING THE DECLARATION AND BODY OF THE MONITOR INSIDE THE PACKAGE BODY AND HAVING EACH OF THE SUBPROGRAMS PROVIDED BY THE PACKAGE SIMPLY CALL THE APPROPRIATE ENTRY OF THE MONITOR.

  THE FIRST APPROACH IS APPROPRIATE IF AN EXISTING PACKAGE DESIGNED FOR USE BY ONE TASK IS TO BE ADAPTED FOR USE BY SEVERAL TASKS. THE SECOND APPROACH IS APPROPRIATE IF THE DATA ABSTRACTION WAS DESIGNED AS A MONITOR IN THE FIRST PLACE. IT PROVIDES AN ORDINARY PACKAGE INTERFACE FOR SHARED DATA.

MONITORS AND PACKAGES

- BOTH CAN BE USED TO ENCAPSULATE DATA STRUCTURES
  - PROVIDE A DATA ABSTRACTION
  - PREVENT THE DATA FROM BEING MANIPULATED EXCEPT BY PRESCRIBED OPERATIONS
  - HIDE IMPLEMENTATION DETAILS
- UNLIKE MONITORS, PACKAGES DO NOT PROTECT AGAINST SIMULTANEOUS UPDATE!
  - SINCE Ada SUBPROGRAMS ARE REENTRANT, TWO TASKS MAY BE SIMULTANEOUSLY EXECUTING SUBPROGRAMS PROVIDED BY THE PACKAGE.
  - IF BOTH TASKS HAVE PASSED THE SAME OBJECT (OR POINTERS TO THE SAME OBJECT) AS PARAMETERS, DIFFERENT OPERATIONS ON THE OBJECT MAY BE INTERLEAVED.
    - *THIS MAY INVALIDATE THE OPERATIONS.*
  - IF THE SUBPROGRAMS MANIPULATE A VARIABLE DECLARED IN THE PACKAGE, THAT VARIABLE IS SUBJECT TO SIMULTANEOUS UPDATE.
- TWO APPROACHES TO MAKING PACKAGES SHAREABLE
  - ENCLOSE THE PACKAGE IN A MONITOR TO PROTECT IT FROM SIMULTANEOUS USE BY MORE THAN ONE TASK.



  - USE A MONITOR TO IMPLEMENT THE PACKAGE

INSTRUCTOR NOTES

THIS SLIDE BEGINS THE SECTION ON MESSAGE BUFFERS. EMPHASIZE BULLET 3.

MESSAGE BUFFERS: RENDEZVOUS AS BUILDING BLOCKS

- RENDEZVOUS ARE THE BASIC MECHANISM BY WHICH TWO Ada TASKS COMMUNICATE.
  - ONE TASK CALLS AN ENTRY OF ANOTHER TASK
  - THE OTHER TASK ACCEPTS A CALL ON THAT ENTRY

- SOMETIMES THIS MECHANISM IS TOO INFLEXIBLE.
  - A TASK SENDING INFORMATION MUST WAIT IF A TASK RECEIVING INFORMATION IS NOT READY FOR A RENDEZVOUS.

- RENDEZVOUS CAN BE USED AS BUILDING BLOCKS TO BUILD MORE POWERFUL COMMUNICATIONS MECHANISMS.
  - SPECIAL NEW TASKS ARE INTRODUCED, AND TWO TASKS COMMUNICATE WITH EACH OTHER INDIRECTLY BY RENDEZVOUSING WITH THESE SPECIAL TASKS.

- <u>MESSAGE BUFFERS</u> ARE SUCH A MECHANISM.
  - THEY ALLOW INFORMATION TO BE SENT BEFORE THE RECIPIENT IS READY TO RECEIVE IT.
  - THE SENDING TASK NEED NOT WAIT FOR THE INFORMATION TO BE RECEIVED.

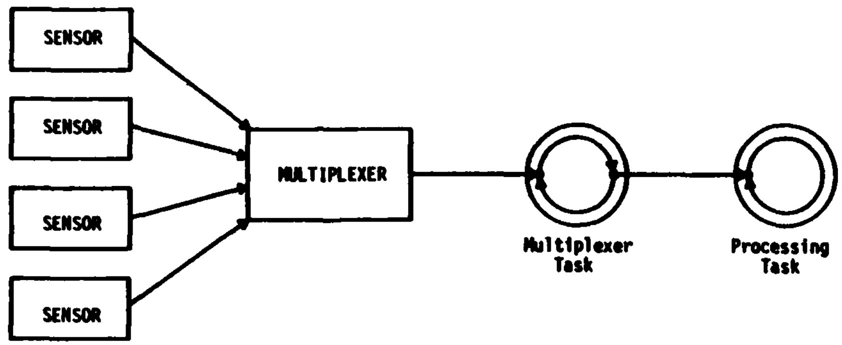
INSTRUCTOR NOTES

THIS IS A SPECIFIC INSTANCE OF A GENERAL PROBLEM:

WHEN ONE TASK PRODUCES DATA THAT ANOTHER TASK CONSUMES, THE CONSUMING TASK MAY FALL BEHIND THE PRODUCING TASK, SLOWING IT DOWN.

- BULLET 2: THE PACKET CONSISTS OF ONE READING FROM EACH SENSOR.
- BULLET 3: THE MULTIPLEXER TASK IS RESPONSIBLE FOR READING FROM THE MULTIPLEXER AT NEARLY EXACT 200 MILLISECOND INTERVALS, SO EACH PACKET ASSEMBLED IS READ ONCE AND ONLY ONCE.
- BULLET 5: THE PROCESSING TASK FALLS BEHIND ONLY OCCASIONALLY, AND ONLY BY A LITTLE BIT, BUT THIS IS ENOUGH TO THROW OFF THE CRITICAL TIMING OF THE MULTIPLEXER TASK.

A COMMON PROBLEM



- SEVERAL SENSORS CONNECTED TO A MULTIPLEXER.
- EVERY 200 MILLISECONDS, THE MULTIPLEXER ASSEMBLES A PACKET OF SENSOR READINGS.
- A MULTIPLEXER TASK READS FROM THE MULTIPLEXER ONCE EVERY 200 MILLISECONDS AND CALLS THE Deliver_Packet ENTRY OF A PROCESSING TASK.
  - CONTENTS OF PACKET PASSED AS A PARAMETER
- THE PROCESSING TASK HAS A LOOP THAT REPEATEDLY ACCEPTS A CALL ON THE Deliver_Packet ENTRY AND PROCESSES THE PACKET DELIVERED.

- THE PROBLEM:
  - THE PROCESSING TASK OCCASIONALLY TAKES A LITTLE MORE THAN 200 MILLISECONDS TO PROCESS A PACKET.
  - MORE THAN 200 MILLISECONDS GO BY WITHOUT A CALL ON Deliver_Packets BEING ACCEPTED, SO THE MULTIPLEXER TASK IS FORCED TO WAIT AT AN ENTRY CALL.
  - SINCE THE MULTIPLEXER TASK GOES MORE THAN 200 MILLISECONDS WITHOUT READING, ONE OF THE PACKETS ASSEMBLED BY THE MULTIPLEXER IS NEVER SEEN.

INSTRUCTOR NOTES

- BULLET 1:

  - ITEM 2: SOMETIMES THE PROCESSING TASK WILL FETCH AN ITEM WAITING IN THE BUFFER AND CONTINUE WITHOUT DELAY. SOMETIMES THE PROCESSING TASK MAY FIND THE BUFFER EMPTY AND BE FORCED TO WAIT.

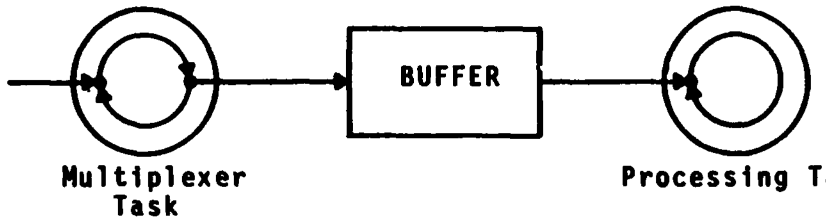
  - ITEM 3: AS LONG AS THE <u>AVERAGE</u> PROCESSING TIME IS UNDER 200 MILLISECONDS AND THE PROCESSING TASK DOES NOT VARY MUCH FROM THIS AVERAGE, THE PROCESSING TASK DOES NOT SLOW DOWN THE MULTIPLEXER TASK.

- BULLET 2:

  REGARDLESS OF IMPLEMENTATION, THE MONITOR HAS TWO OPERATIONS (ENTRIES): PLACING AN ELEMENT IN THE BUFFER AND REMOVING THE OLDEST ELEMENT FROM THE BUFFER.

  - ITEM 3: THIS IS OVERKILL BECAUSE: (1) IF THE PRODUCING TASK WORKS FASTER ON THE AVERAGE THAN THE CONSUMING TASK, EVEN AN UNBOUNDED QUEUE WILL NOT HELP; THE CONSUMING TASK CAN NEVER CATCH UP, SO THERE IS A SERIOUS FLAW IN SYSTEM DESIGN. (2) IF THE PRODUCING TASK WORKS MORE SLOWLY ON THE AVERAGE THAN THE CONSUMING TASK, THE PROBABILITY OF A FIXED-SIZE n-ELEMENT BUFFER BECOMING FULL DECREASES EXPONENTIALLY.

SOLUTION: MESSAGE BUFFERS



- WHAT IT DOES:
  - ACCEPTS PACKETS FROM THE MULTIPLEXER TASK AT 200 MILLISECOND INTERVALS.
  - DELIVERS PACKETS TO PROCESSING TASK UPON REQUEST IN ORDER OF RECEIPT.
  - MULTIPLEXER TASK AND PROCESSING TASK NO LONGER NEED BE CLOSELY SYNCHRONIZED.

- MESSAGE BUFFER CAN BE IMPLEMENTED AS A MONITOR WITH OPERATIONS Send AND Receive.
  - ONE-ELEMENT MESSAGE BUFFER:
    - MONITOR PROTECTS A VARIABLE THAT CAN HOLD ONE PACKET.
    - MULTIPLEXER TASK CAN GET ONE PACKET AHEAD OF PROCESSING TASK.
  - n-ELEMENT MESSAGE BUFFER:
    - MONITOR PROTECTS A QUEUE OF UP TO n PACKETS.
    - MULTIPLEXER TASK CAN GET n PACKETS AHEAD OF PROCESSING TASK.
  - UNBOUNDED MESSAGE BUFFER:
    - MONITOR PROTECTS A DYNAMICALLY ALLOCATED LINKED LIST.
    - MULTIPLEXER TASK CAN GET ARBITRARILY FAR AHEAD OF PROCESSING TASK.
    - THIS IS USUALLY OVERKILL.

INSTRUCTOR NOTES

THIS SECTION BEGAN BY EXPLAINING THAT RENDEZVOUS CAN BE USED AS BUILDING BLOCKS TO CONSTRUCT OTHER COMMUNICATIONS MECHANISMS.

THE SECTION CONCLUDES BY POINTING OUT THAT RENDEZVOUS ARE NOT THE ONLY WAY TO DO THIS. IF NECESSARY, MESSAGE BUFFERS CAN BE IMPLEMENTED DIRECTLY IN HARDWARE (PERHAPS USING A TEST-AND-SET MACHINE INSTRUCTION TO ENSURE MUTUAL EXCLUSION).

EMPHASIZE THAT WE DO NOT MEAN TO SUGGEST THAT RENDEZVOUS ARE <u>NECESSARILY</u> INEFFICIENT.

VERY FAST MESSAGE BUFFERS

- SOME PROGRAM DESIGNS USE MESSAGE BUFFERS FOR ALL INTERTASK COMMUNICATION.
- EFFICIENT SERVICING OF Send AND Receive OPERATIONS MAY BE CRITICAL.
- HARDWARE MAY SUPPORT A MORE EFFICIENT IMPLEMENTATION THAN MONITORS.
- Send AND Receive OPERATIONS CAN BE IMPLEMENTED IN MACHINE CODE.
  - Send AND Receive PROCEDURES DECLARED IN A PACKAGE.
  - CODE PROCEDURE HIDDEN IN PACKAGE BODY.
  - MUTUAL EXCLUSION PROVIDED BY HARDWARE OR RUNTIME SYSTEM.
- NOT RECOMMENDED AS THE USUAL PRACTICE, BUT THE OPTION IS AVAILABLE WHEN NEEDED.

INSTRUCTOR NOTES

- BULLET 2: A TRANSFORMATION WITH ZERO INPUT STREAMS COULD BE ONE PRODUCING PULSES EVERY 100 MILLISECONDS OR ONE PRODUCING A STREAM OF BOOLEAN VALUES, WHERE THE n^{th} VALUE IN THE STREAM IS TRUE IF AND ONLY IF $n \bmod 3 = 0$.

- BULLET 4: THE SECOND AND THIRD SUBBULLETS EXPLAIN WHY STREAM-ORIENTED TASK DESIGN REDUCES COMPLEXITY. IT DECOMPOSES A COMPLEX PROBLEM INTO SEVERAL <u>INDEPENDENT</u> SIMPLE SUBPROBLEMS. IT ALSO SEPARATES CONCERN WITH THE DECOMPOSITION FROM CONCERN WITH THE SOLUTIONS TO THE SUBPROBLEMS.

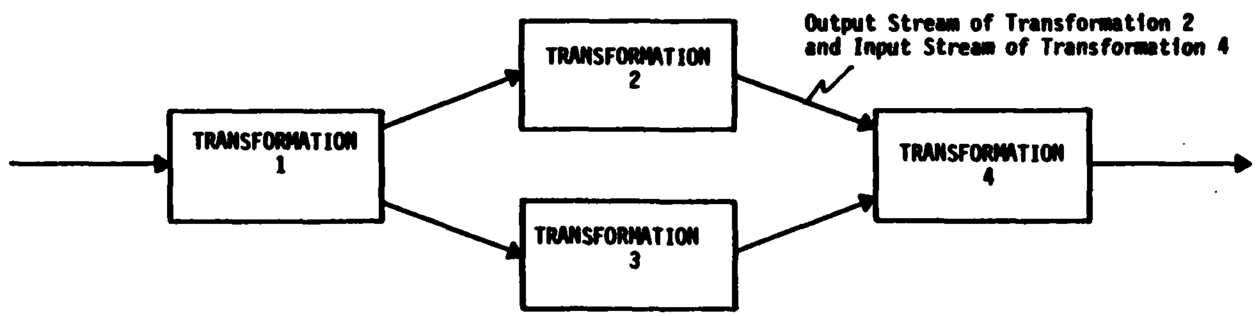
STREAM-ORIENTED TASK DESIGN

- SOME PROBLEMS ARE MOST EASILY UNDERSTOOD AS A COLLECTION OF TRANSFORMATIONS ON STREAMS OF DATA.

- EACH TRANSFORMATION HAS ZERO OR MORE INPUT STREAMS AND ONE OR MORE OUTPUT STREAMS.

- EACH TRANSFORMATION IS SIMPLE
  - OBTAIN DATA ITEMS IN ORDER FROM THE INPUT STREAMS.
  - SEND DATA ITEMS IN ORDER TO THE OUTPUT STREAMS.
  - STRAIGHTFORWARD WAY TO DERIVE OUTPUT ITEMS FROM INPUT ITEMS.

- BY CONNECTING SIMPLE TRANSFORMATIONS WE PRODUCE MORE COMPLEX TRANSFORMATIONS.
  - ONE TRANSFORMATION'S OUTPUT STREAM SERVES AS ANOTHER TRANSFORMATION INPUT STREAM.



  - INTERCONNECTION TREATS TRANSFORMATIONS AS "BLACK BOXES."
  - TRANSFORMATIONS WRITTEN WITHOUT CONCERN FOR THE INTERCONNECTIONS.

INSTRUCTOR NOTES

- BULLET 1: JSP AND JSD ARE SOFTWARE ENGINEERING METHODOLOGIES BASED ON DATA STREAMS AND TRANSFORMATIONS.

  REFER STUDENTS TO THE JACKSON REFERENCE IN THE BIBLIOGRAPHY.

  AS WE SHALL SEE LATER, IT IS EASY TO IMPLEMENT STREAM-ORIENTED DESIGNS WITH Ada TASKS. THIS MAKES THE JACKSON METHODOLOGY EXTREMELY SUITABLE FOR USE WITH Ada.

- BULLET 2: BECAUSE THE FILES ARE SEQUENTIAL, EACH FILE ELEMENT IS A LINE. THUS FILES MUST BE READ AND WRITTEN LINE-BY-LINE.

  ANOTHER EXAMPLE OF A STRUCTURE CLASH IS IN THE MESSAGE COMPARISON PROBLEM. IN THIS CASE THE STRUCTURES OF THE INPUT STREAMS CLASH BECAUSE MESSAGES CAN BE BROKEN INTO BLOCKS IN DIFFERENT WAYS IN EACH STREAM.

  IT IS FOR PROBLEMS WITH STRUCTURE CLASHES THAT STREAM-ORIENTED DESIGN IS MOST USEFUL.

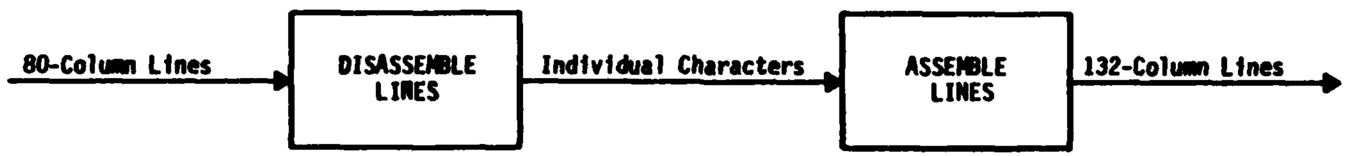
- BULLET 3: THE REFORMATTING IS BROKEN INTO TWO VERY SIMPLE TRANSFORMATIONS. THE FIRST TAKES AN INPUT STREAM OF 80-COLUMN LINES AND PRODUCES AN OUTPUT STREAM CONSISTING OF THE INDIVIDUAL CHARACTERS IN THOSE LINES.

  THE SECOND TAKES AN INPUT STREAM OF INDIVIDUAL CHARACTERS, GROUPS THEM INTO 132-CHARACTER LINES, AND PRINTS THE LINES.

- BULLET 4: DATA IS PASSED THROUGH AN *in* PARAMETER OF THE ENTRY CALL. IT IS ALSO POSSIBLE FOR THE RENDEZVOUS TO GO IN THE OPPOSITE DIRECTION (RECEIVING TASK CALLS ON ENTRY OF THE SENDING TASK TO OBTAIN DATA) WITH DATA PASSED THROUGH AN *out* PARAMETER.

STRUCTURE CLASHES

- JACKSON STRUCTURED PROGRAMMING (JSP) AND JACKSON SYSTEM DEVELOPMENT (JSD) BASE PROGRAM STRUCTURE ON THE STRUCTURE OF INPUT OR OUTPUT STREAMS.
  - PROGRAMS EASY TO WRITE WHEN STRUCTURES MATCH.
  - PROGRAMS HARD TO WRITE WHEN STRUCTURES CLASH.
- EXAMPLE OF A STRUCTURE CLASH:
  - PROGRAM TO REFORMAT A FILE OF 80-COLUMN LINES INTO A FILE OF 132-CHARACTER LINES, CHARACTER-FOR-CHARACTER.
  - INPUT STRUCTURE: 80-COLUMN LINES
  - OUTPUT STRUCTURE: 132-CHARACTER LINES
- STREAM TRANSFORMATIONS CAN BE USED TO RESOLVE STRUCTURE CLASHES:



- IN ADA, TRANSFORMATIONS ARE IMPLEMENTED AS TASKS.
  - A TASK SENDS DATA TO AN OUTPUT STREAM BY CALLING AN ENTRY OF THE TASK AT THE OTHER END OF THE STREAM.
  - A TASK RECEIVES DATA FROM AN INPUT STREAM BY ACCEPTING THE CORRESPONDING ENTRY.
  - DATA STREAMS ARE REALLY SERIES OF RENDEZVOUS.

INSTRUCTOR NOTES

- BULLET 3:

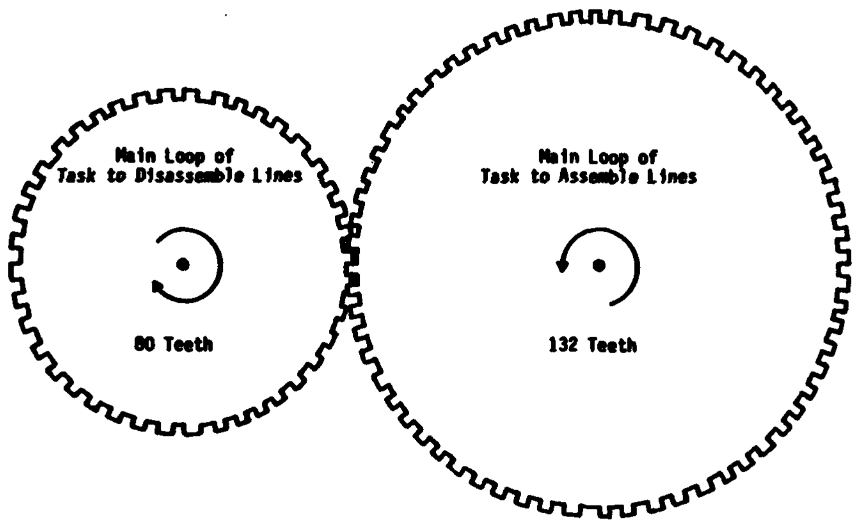
  ONE ENTRY CALL IS MADE FOR EACH ONE ACCEPTED, SO THE TWO TASKS ADVANCE IN STEP WITH EACH OTHER ON THE CHARACTER (TOOTH) LEVEL.

- BULLET 4 and 5:

  THE GEARS TURN AT DIFFERENT RATES. THIS REFLECTS THE FACT THAT THE TASKS OPERATE AT DIFFERENT RATES ON THE LINE LEVEL. THIS IS THE SOURCE OF DIFFICULTY IN A PROBLEM WITH STRUCTURE CLASHES. STREAM-ORIENTED DESIGN ALLEVIATES THE DIFFICULTY BY ALLOWING EACH GEAR TO VIEW THE OTHER AS AN INFINITE STREAM OF TEETH RATHER THAN AS ANOTHER GEAR WITH ITS OWN ROTATION RATE.

STREAM-ORIENTED TASKS ARE TIGHTLY SYNCHRONIZED

- TASK TO DISASSEMBLE LINES MAKES 80 ENTRY CALLS FOR EACH INPUT LINE DISASSEMBLED (ONE CALL FOR EACH CHARACTER).
- TASK TO ASSEMBLE LINES ACCEPTS 132 ENTRY CALLS FOR EACH OUTPUT LINE ASSEMBLED (ONE CALL FOR EACH CHARACTER).



- EACH TOOTH REPRESENTS ONE RENDEZVOUS.
- A FULL ROTATION OF THE LEFT GEAR REPRESENTS COMPLETE PROCESSING OF AN INPUT LINE.
- A FULL ROTATION OF THE RIGHT GEAR REPRESENTS COMPLETE PROCESSING OF AN OUTPUT LINE.

INSTRUCTOR NOTES

THE NEXT GENERATION OF PROGRAMMING LANGUAGES AND MACHINES MAY BE BASED ON THE DATA FLOW MODEL. WRITING DATA FLOW Ada PROGRAMS MAY MAKE THESE PROGRAMS MORE ADAPTABLE IN THE FUTURE.

- BULLET 2:

  IN A DATA FLOW COMPUTATION, ACTIONS ARE NOT ORDERED BY AN INSTRUCTION COUNTER, BUT BY THE AVAILABILITY OF DATA. A TRANSFORMATION "FIRES" WHEN ITS INPUTS ARE AVAILABLE, AND GENERATES AN OUTPUT. THIS OUTPUT MAY IN TURN CAUSE ANOTHER TRANSFORMATION TO FIRE.

  IN THE TEXT REFORMATTING PROBLEM, WE HAVE STARTED WITH FAIRLY HIGH-LEVEL TRANSFORMATIONS, IMPLEMENTED USING von NEUMAN-STYLE Ada. IN A PURE DATA FLOW LANGUAGE, EVEN ASSEMBLY AND DISASSEMBLY OF LINES WOULD BE SPECIFIED IN TERMS OF MORE PRIMITIVE TRANSFORMATIONS.

- BULLETS 3 AND 4:

  IF STUDENTS ARE INTERESTED IN DETAILS, REFER THEM TO THE REFERENCE IN BULLET 6, PARTICULARLY THE INTRODUCTORY ARTICLE.

- BULLET 5:

  ACOS (ASP COMMON OPERATIONAL SOFTWARE) IS A DATA FLOW METHODOLOGY FOR PROGRAMMING THE NAVY'S AN/UYS-1 ADVANCED SIGNAL PROCESSOR (ASP). ECOS (EMSP COMMON OPERATIONAL SOFTWARE) IS A SIMILAR METHODOLOGY FOR PROGRAMMING THE NAVY'S ENHANCED MODULAR SIGNAL PROCESSOR (EMSP).

  THE IMPORTANT POINT TO BE MADE HERE IS THAT DATA FLOW PROGRAMMING IS NOT A WILD ACADEMIC PIPE DREAM. IT IS APPLICABLE TO EMBEDDED MILITARY APPLICATIONS.

- BULLET 6:

  THIS REFERENCE IS IN THE BIBLIOGRAPHY.

DATA FLOW PROGRAMMING

- DATA FLOW PROGRAMMING IS A STYLE OF PROGRAMMING BASED ON DATA STREAMS AND TRANSFORMATIONS.
- IT IS BASED ON A DIFFERENT VIEW OF THE NATURE OF COMPUTATION.
  - THE TRADITIONAL, OR VON NEUMANN MODEL:
    - GLOBAL, ADDRESSABLE MEMORY THAT IS FREQUENTLY UPDATED.
    - INSTRUCTIONS EXECUTED IN FIXED SEQUENCE, USING AN INSTRUCTION COUNTER.
  - THE DATA FLOW MODEL:
    - NO NAMED CONTAINERS (VARIABLES), ONLY VALUES
    - TRANSFORMATIONS ARE SIMPLE OPERATIONS, E.G. +, -, *, /
    - WHEN AN INPUT VALUE IS AVAILABLE FROM EACH INPUT STREAM, THE TRANSFORMATION COMPUTES A RESULT VALUE AND PLACES IT IN THE OUTPUT STREAM.
    - DATA FLOW "PROGRAMS" ARE JUST STREAM/TRANSFORMATION DIAGRAMS.
- DATA FLOW COMPUTERS HAVE BEEN DESIGNED AND BUILT.
- PROGRAMMING LANGUAGES HAVE BEEN DESIGNED AROUND THE DATA FLOW MODEL OF COMPUTATION.
- DATA FLOW METHODS HAVE BEEN APPLIED TO EMBEDDED SIGNAL-PROCESSING APPLICATIONS (ACOS, ECOS).
- FEBRUARY 1982 ISSUE OF IEEE <u>COMPUTER</u> IS DEVOTED TO DATA FLOW SYSTEMS.

INSTRUCTOR NOTES

- BULLET 2: THIS DOES NOT MEAN THAT PERFORMANCE <u>WILL</u> BE ADVERSELY AFFECTED BY A LARGE NUMBER OF TASKS, ONLY THAT FOR <u>SOME</u> IMPLEMENTATIONS IT <u>MIGHT</u> BE.

- BULLET 3: EMPHASIZE THE "IF".

  CALL INVERSION A "MANIPULATION" RATHER THAN A "TRANSFORMATION" SINCE THIS SECTION USES THE TERM "TRANSFORMATION" TO MEAN AN ENTITY THAT MAPS INPUT STREAMS TO OUTPUT STREAMS.

- BULLET 4: EACH IMPLEMENTATION CAN DEFINE ITS OWN PROGRAMS TO SUPPLEMENT THE LANGUAGE-DEFINED PRAGMAS. THE HILFINGER REFERENCE IN THE BIBLIOGRAPHY PROPOSES SUCH PRAGMAS.

  AN UNPUBLISHED PAPER BY NASSI AND HABERMANN DESCRIBES CIRCUMSTANCES IN WHICH A COMPILER CAN MAKE A SIMILAR OPTIMIZATION WITHOUT AID OF A PRAGMA.

- BULLET 5: THE STREAM/TRANSFORMATION MODEL CAN HELP DESIGNERS AND PROGRAMMERS TO UNDERSTAND A COMPLEX PROBLEM. GIVEN A STREAM-ORIENTED SOLUTION, NO CREATIVE THOUGHT IS NECESSARY TO MERGE TASKS - JUST FOLLOW THE RECIPE.

  EMPHASIZE THE "IF" IN THE SECOND ITEM.

STREAM-ORIENTED TASK DESIGNS AND EFFICIENCY

- STREAM-ORIENTED TASK DESIGN DECOMPOSES A SEQUENTIAL PROBLEM INTO A POTENTIALLY LARGE NUMBER OF TASKS.
- FOR SOME Ada IMPLEMENTATIONS, THIS MAY SERIOUSLY AFFECT PERFORMANCE.
- IF THIS TURNS OUT TO BE A PROBLEM, THERE ARE MECHANICAL MANIPULATIONS FOR MERGING A SERIES OF TRANSFORMATIONS INTO A SINGLE TASK.
    - THIS MANIPULATION IS ESSENTIALLY JACKSON'S "PROGRAM INVERSION"
- SOME COMPILERS MAY BE CLEVER ENOUGH TO PERFORM THIS MANIPULATION INTERNALLY.
    - PRAGMAS COULD ALERT THE COMPILER TO THE FACT THAT CERTAIN TASKS CAN BE MERGED.
    - THESE TASKS WILL BE COMPILED AS COROUTINES, WITHOUT EXPENSIVE "CONTEXT SWITCHES" WHEN ONE TASK PASSES CONTROL TO ANOTHER.
    - ONLY THE OBJECT CODE WOULD BE AFFECTED. THE SIMPLICITY AND CLARITY OF THE SOURCE CODE WOULD BE PRESERVED.
- WHETHER OR NOT TASKS MUST BE MERGED TO MEET PERFORMANCE REQUIREMENTS, STREAM-ORIENTED TASK DESIGN IS USEFUL AS A <u>DESIGN TOOL</u>.
    - FIRST WRITE A CLEAR, SIMPLE, BUT POTENTIALLY INEFFICIENT SOLUTION.
    - IF NECESSARY, MECHANICALLY MANIPULATE THIS SOLUTION TO OBTAIN A LESS CLEAR BUT MORE EFFICIENT SOLUTION.

INSTRUCTOR NOTES

- BULLET 2:

A DETAILED EXAMINATION OF THESE CONSIDERATIONS IS BEYOND THE SCOPE OF THIS MODULE. THE IMPORTANT POINT IS THAT MINIMUM FREQUENCY REQUIREMENTS ARE EXTERNALLY IMPOSED, NOT SUBJECT TO THE DISCRETION OF THE SOFTWARE DESIGNER OR PROGRAMMER.

- BULLET 3:

THE NEXT TWO SLIDES DESCRIBE TRADITIONAL CYCLIC EXECUTIVES.

CYCLIC PROCESSING

- IN TYPICAL REAL-TIME APPLICATIONS (E.G. AVIONICS AND GUIDANCE), CERTAIN ACTIONS MUST BE PERFORMED REPETITIVELY, AT SPECIFIED INTERVALS.
  - DATA SAMPLING
  - CONTROL (FEEDBACK) LOOPS

- INTERVALS ARE BASED ON THE NATURE OF THE APPLICATION AND SYSTEMS CONTROL THEORY.
  - IF DATA BEING SAMPLED FLUCTUATES AT SOME FREQUENCY, IT SHOULD BE SAMPLED AT AT LEAST TWICE THAT FREQUENCY TO AVOID ERRONEOUS EXTRAPOLATIONS (ALIASING).
  - "TRANSPORT LAG" BETWEEN FEEDBACK AND OUTPUT IN A CONTROL LOOP MUST BE KEPT SMALL, OR FEEDBACK WILL BE OUT OF PHASE. CONTROL LOOP CAN BECOME UNSTABLE.

- TYPICAL SOLUTION IS A CYCLIC EXECUTIVE.
  - PROCESSING "TASKS" ACTIVATED IN A FIXED ORDER AT A FIXED FREQUENCY.
  - THESE "TASKS" ARE SMALL NON-CONCURRENT PROCESSING STEPS.
  - WE'LL CALL THESE "TASKS" <u>ACTIVITIES</u> TO AVOID CONFUSION.
  - SCHEDULING OF ACTIVITIES BASED ON "MAJOR CYCLES" AND "MINOR CYCLES"

INSTRUCTOR NOTES

- BULLET 1: THESE ARE ASSUMED EXTERNAL FREQUENCY REQUIREMENTS.

- BULLET 2: THE <u>MINOR CYCLE</u> IS A LOOP EXECUTED AT REGULAR INTERVALS SMALL ENOUGH TO ACCOMMODATE THE HIGHEST REQUIRED PROCESSING FREQUENCY. IN THIS EXAMPLE, THE MINOR CYCLE IS EXECUTED 250 TIMES A SECOND (ONCE EVERY 4 MILLISECONDS) TO MEET THE REQUIREMENTS OF ACTIVITY A. ACTIVITIES TO BE PERFORMED AT LOWER FREQUENCIES ARE PERFORMED EVERY 1, 2, 4, 8, ... MINOR CYCLES. THE LARGEST POWER OF TWO THAT RESULTS IN A SUFFICIENTLY HIGH FREQUENCY IS CHOSEN.

  SOMETIMES AN ACTIVITY IS TOO LONG TO FIT INTO ONE MINOR CYCLE AND MUST BE SPLIT INTO PIECES EXECUTED IN SUCCESSIVE MINOR CYCLES.

  <u>NOTE</u>: AN ACTIVITY PERFORMED (FOR EXAMPLE) 62.5 TIMES A SECOND IS SOMETIMES DESCRIBED AS OPERATING AT 62.5 HERTZ (62.5 Hz).

- BULLET 3: THE <u>MAJOR CYCLE</u> CONSISTS OF SOME PREDETERMINED NUMBER OF MINOR CYCLES. EACH MAJOR CYCLE CONSISTS OF THE SAME SEQUENCE OF ACTIONS, AND PROGRAM EXECUTION CONSISTS OF REPEATED EXECUTION OF THE MAJOR CYCLE. THE SIZE OF THE MAJOR CYCLE IS BASED ON THE IMPLEMENTED FREQUENCY USED BY THE LOWEST FREQUENCY ACTIVITY. IN THIS EXAMPLE, ACTIVITY E IS PERFORMED ONCE EVERY 8 MINOR CYCLES, SO THERE ARE EIGHT MINOR CYCLES PER MAJOR CYCLE.

- BULLET 4: THE BOTTOM ROW SHOWS THE DIVISION OF THE MAJOR CYCLE INTO 8 MINOR CYCLES EACH LASTING 4 MILLISECONDS. DIFFERENT COMBINATIONS OF ACTIVITIES ARE SCHEDULED FOR EACH MINOR CYCLE. ALL PROCESSING IS SEQUENTIAL. (IN THE FIRST MINOR CYCLE, FOR EXAMPLE, ACTIVITY A IS EXECUTED, THEN ACTIVITY B, THEN ACTIVITY E. THEN THE SYSTEM WAITS UNTIL IT IS TIME FOR THE NEXT MINOR CYCLE.)

  THE FOUR ROWS LABELED "REQUIRED PROCESSING" SHOW ACTIVITIES THAT MUST BE PERFORMED ONCE WITHIN SPECIFIED SLICES OF THE MAJOR CYCLE. ALLOCATION OF AN ACTIVITY TO A PARTICULAR MINOR CYCLE WITHIN A SLICE IS DISCUSSED ON THE NEXT SLIDE.

EXAMPLE OF MAJOR AND MINOR CYCLES

- ACTIVITIES AND MINIMUM ALLOWABLE FREQUENCIES:

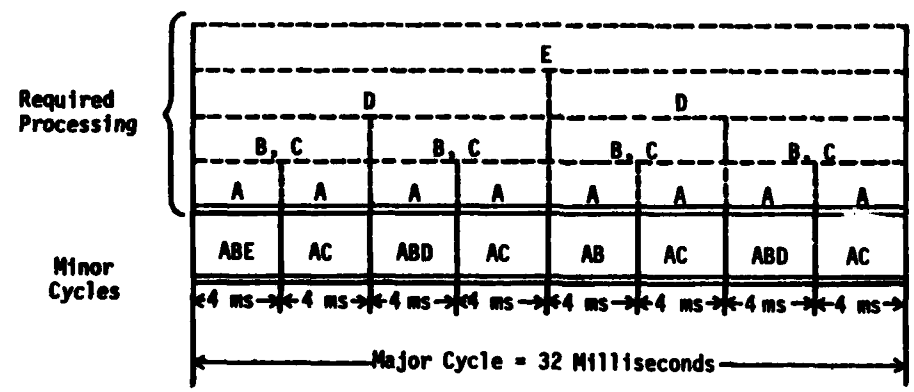
  ACTIVITY A: AT LEAST 250 TIMES A SECOND
  ACTIVITY B: AT LEAST 100 TIMES A SECOND
  ACTIVITY C: AT LEAST 75 TIMES A SECOND
  ACTIVITY D: AT LEAST 50 TIMES A SECOND
  ACTIVITY E: AT LEAST 30 TIMES A SECOND

- FREQUENCIES BASED ON 4 MILLISECOND <u>MINOR CYCLE TIME</u>:

  ACTIVITY A: ONCE EVERY MINOR CYCLE (250 TIMES A SECOND)
  ACTIVITY B: ONCE EVERY 2 MINOR CYCLES (125 TIMES A SECOND)
  ACTIVITY C: ONCE EVERY 2 MINOR CYCLES (125 TIMES A SECOND)
  ACTIVITY D: ONCE EVERY 4 MINOR CYCLES (62.5 TIMES A SECOND)
  ACTIVITY E: ONCE EVERY 8 MINOR CYCLES (31.25 TIMES A SECOND)

- <u>MAJOR CYCLE</u> CONSISTS OF 8 MINOR CYCLES.

- POSSIBLE SCHEDULING OF ACTIVITIES WITHIN A MAJOR CYCLE:



INSTRUCTOR NOTES

- BULLET 1: THIS ALLOCATION TAKES PLACE AT SYSTEM DESIGN TIME.
- BULLET 2:
  - ITEM 1: AN ACTIVITY SHOULD BE SCHEDULED IN A SUFFICIENT NUMBER OF MINOR CYCLES.
  - ITEM 2: THE MINOR CYCLES IN WHICH AN ACTIVITY IS SCHEDULED SHOULD BE EVENLY SPACED. AN ACTIVITY SHOULD NOT BE SCHEDULED TO RUN IN TWO CONSECUTIVE MINOR CYCLES AND REMAIN PASSIVE IN THE NEXT TWO.
  - ITEM 3: PROCESSING SHOULD NOT BE CROWDED INTO A FEW MINOR CYCLES, BUT DISTRIBUTED EVENLY OVER ALL MINOR CYCLES.
  -ITEM 4: AN ACTIVITY SETTING CERTAIN VARIABLES MAY HAVE TO BE SCHEDULED BEFORE AN ACTIVITY EXAMINING THOSE VARIABLES.
- BULLET 3: IN PRACTICE, THE PURE CYCLIC EXECUTIVE WE HAVE DESCRIBED IS ONLY USEFUL FOR VERY SIMPLE SYSTEMS.
  - ITEM 1: IN AVIONICS SYSTEMS, FOR EXAMPLE, "BURST" COMPUTING LOADS, FAULT RECOVERY, AND COMMAND PROCESSING ARE NOT PERIODIC. RADAR TRACKING AND COMMUNICATIONS SYSTEMS ARE PRIMARILY DRIVEN BY ASYNCHRONOUS EVENTS.
  - ITEM 2: MORE COMPLICATED EXECUTIVES ARE REQUIRED TO HANDLE VARIATIONS IN TIMING.
  - ITEM 3: AS LONG AS ALL ACTIVITIES ACCESSED GLOBAL DATA IN A FIXED, WELL-UNDERSTOOD ORDER, THERE WAS NO DANGER OF SIMULTANEOUS UPDATE. ASYNCHRONOUS PROCESSING INTRODUCES THE POSSIBILITY OF INTERLEAVED ACCESS TO DATA.

SCHEDULING OF ACTIVITIES WITHIN THE MAJOR CYCLE

- MANY POSSIBILITIES. ALLOCATION OF ACTIVITIES TO SPECIFIC MINOR CYCLES IS A DIFFICULT MANUAL ACTIVITY.

- CRITERIA:
  - REQUIRED FREQUENCY
  - REGULARITY OF INTERVALS (BCBC VERSUS BBCC)
  - DISTRIBUTION OF PROCESSING LOAD
  - DATA DEPENDENCY
    - ACTIVITIES COMMUNICATE BY SETTING AND EXAMINING SHARED (GLOBAL) VARIABLES.

- COMPLICATIONS:
  - CERTAIN ACTIVITIES MAY NOT BE PERIODIC
    (EVENT-DRIVEN PROCESSING, BURST LOADS, BACKGROUND PROCESSING)
  - RESULTING ASYNCHRONISM COMPLICATES SCHEDULING
    (CYCLE OVERFLOW, DYNAMIC RESCHEDULING, BUFFERING)
  - DEPARTURE FROM PREDETERMINED ORDER INTRODUCES DANGER OF SIMULTANEOUS UPDATE OF SHARED VARIABLES.

INSTRUCTOR NOTES

- BULLET 1: THE DIRECT IMPLEMENTATION INVOLVES A SINGLE TASK EXECUTING A LOOP OF THE FOLLOWING FORM:

```
loop
   accept Timer_Interrupt;
   Minor_Cycle_Number :=
         (Minor_Cycle_Number mod Number_of_Minor_Cycles) + 1;
   case Minor_Cycle_Number is
      ...
   end case;
end loop;
```

  SOME FEEL THAT THIS APPROACH IS APPROPRIATE, BECAUSE OF ITS FAMILIARITY, EFFICIENCY, AND SIMPLICITY, FOR SYSTEMS SIMPLE ENOUGH TO USE A PURE CYCLIC EXECUTIVE.

- BULLET 2: THE MODEL DESCRIBED HERE BECOMES MORE APPROPRIATE AS SYSTEMS BECOME MORE ASYNCHRONOUS AND SCHEDULING BECOMES MORE COMPLEX.

  - ITEM 3: RENDEZVOUS ARE USED TO ENSURE THAT DIFFERENT TASKS, REPEATING DIFFERENT ACTIVITIES AT APPROPRIATE FREQUENCIES, PERFORM THOSE ACTIVITIES IN THE RIGHT ORDER. THIS IS DEALT WITH IN MORE DETAIL TWO SLIDES FROM NOW.

  - ITEM 4: IN A CYCLIC EXECUTIVE, ACTIVITIES ARE OFTEN BROKEN UP INTO PIECES THAT WILL FIT INTO MINOR CYCLES. IN THE APPROACH DESCRIBED HERE, ACTIONS THAT ARE CONCEPTUALLY SUCCESSIVE STEPS OF THE SAME ACTIVITY GO IN THE SAME LOOP.

- BULLET 3: THE TASK REPEATEDLY PERFORMS Activity_X AND DELAYS FOR THE AMOUNT OF TIME REQUIRED TO REPEAT WITH FREQUENCY f. THE METHOD OF COMPUTING THE DELAY (WHICH MAY VARY FROM ONE ITERATION TO ANOTHER) IS DESCRIBED ON THE NEXT SLIDE). WE AIM EXACTLY FOR FREQUENCY f, NOT A MINOR CYCLE FREQUENCY DIVIDED BY SOME POWER OF TWO.

CYCLIC PROCESSING IN Ada

- Ada DOESN'T PRECLUDE THE TRADITIONAL APPROACH.
  - CYCLIC SCHEDULING, DRIVEN BY TIMER INTERRUPTS, CAN BE IMPLEMENTED EASILY IN Ada.
  - THIS DOES NOT TAKE ADVANTAGE OF Ada's STRENGTHS.

- THE PREFERRED Ada APPROACH INVOLVES ONE TASK FOR EACH INDEPENDENT ACTIVITY TO BE PERFORMED REPETITIVELY.
  - EACH TASK EXECUTES A LOOP AT THE REQUIRED FREQUENCY.
  - ONE ITERATION EXECUTES A DELAY STATEMENT THAT EXPIRES IN TIME FOR THE NEXT ITERATION.
  - RENDEZVOUS ARE USED FOR COMMUNICATION AND SYNCHRONIZATION AMONG TASKS.
  - EACH TASK CORRESPONDS TO A SINGLE CONCEPTUAL THREAD.

- SUPPOSE Activity_X IS AN ACTIVITY TO BE PERFORMED REPEATEDLY WITH A FREQUENCY OF _f_:

```
task body Activity_X_Task is
   ...
begin
   ...
   loop
      Activity_X;
      delay [an amount of time required to achieve frequency f];
   end loop;
end Activity_X_Task;
```

- BULLET 2:

  ASSUME Next_Iteration_Time IS THE TIME AT WHICH THE NEXT ITERATION IS SCHEDULED TO START. SINCE THE FUNCTION CALL CLOCK RETURNS THE CURRENT TIME, Next_Iteration_Time_Clock IS THE AMOUNT OF TIME FROM "NOW" UNTIL THE NEXT SCHEDULED ITERATION.

- BULLET 3:

  THE CONTEXT CLAUSE "with CALENDAR; use CALENDAR;" IS ASSUMED.

  THIS APPROACH DOES NOT FORCE EACH ITERATION TO START ON TIME, BUT IT TENDS TO LIMIT THE LAG BETWEEN SCHEDULED AND ACTUAL START OF EACH ITERATION.

- BULLET 4:

  - ITEM 3: CUMULATIVE DRIFT MEANS THAT THE TASK FALLS FURTHER AND FURTHER BEHIND AND NEVER CATCHES UP. IT IS HARMFUL BECAUSE (1) THE ACTIVITY SLIPS OUT OF PHASE WITH THE OTHER CYCLIC PROCESSING IN THE SYSTEM AND (2) IN THE LONG RUN, THE ACTIVITY IS EXECUTED AT LESS THAN THE REQUIRED FREQUENCY. JITTER MEANS THAT AN ACTIVITY IS REPEATED AT PRECISELY THE RIGHT FREQUENCY IN THE LONG RUN, BUT WITH LOCAL VARIATIONS. ITERATION TIMES MAY NOT BE EVENLY SPACED, BUT THE DEVIATION OF EACH ITERATION TIME FROM THE SCHEDULED TIME IS INDEPENDENT; THERE IS NO CUMULATIVE EFFECT.

ACHIEVING DESIRED CYCLE FREQUENCIES

- KEEP TRACK OF THE <u>EXACT</u> TIME THE LOOP IS <u>SCHEDULED</u> TO PERFORM THE NEXT ITERATION.

- DELAY ONLY UNTIL THEN:

```
delay Next_Iteration_Time - Clock;
```

- RESULTING TASK BODY:

```
task body A_Task is
   Cycle_Duration : Constant := 0.004;
      -- Activity_A should be performed once every 4 milliseconds
   ...
begin
   ...
   Next_Iteration_Time := Clock;
   loop
      Activity_A;
      Next_Iteration_Time := Next_Iteration_Time + Cycle_Duration;
      delay Next_Iteration_Time - Clock;
   end loop;
end A_Task;
```

- THIS APPROACH IS NECESSARY BECAUSE DELAY STATEMENTS CAUSE A TASK TO PAUSE FOR <u>AT LEAST</u> THE SPECIFIED DURATION.
  - PROCESSOR MIGHT NOT BE ALLOCATED TO A TASK THE MOMENT ITS DELAY EXPIRES.
  - WE COMPENSATE FOR THIS BY MAKING THE NEXT DELAY SHORTER.
  - ACTUAL ITERATION TIMES MAY EXHIBIT "JITTER" BUT NOT <u>CUMULATIVE</u> DRIFT.

- BULLET 3:

  BY ACCEPTING THE CALL ON Get_Data, Activity_1_Task "OPENS THE GATE" TO LET Activity_2_Task THROUGH.

  THE "GATE" COULD ALSO HAVE BEEN IMPLEMENTED WITH THE ENTRY CALL GOING IN THE OTHER DIRECTION.

  THE TIME TO EXECUTE Activity_1 SHOULD BE A SMALL FRACTION OF THE Activity_1_Task CYCLE TIME. SIMILAR CONDITIONS SHOULD HOLD FOR OTHER CYCLIC TASKS. THIS IS WHAT IS MEANT BY "REASONABLE PROCESSOR LOAD."

SYNCHRONIZING ACTIVITIES

- SOMETIMES ACTIVITY 1 MUST EXECUTE BEFORE ACTIVITY 2 BECAUSE ACTIVITY 1 SETS VARIABLES OR CREATES OTHER CONDITIONS THAT ACTIVITY 2 DEPENDS ON.

- WITH SINGLE-THREAD TASKS, TIMING OF ACTIVITY 2'S LOOP CAN BE CONTROLLED BY A RENDEZVOUS INSTEAD OF A DELAY STATEMENT. DATA CAN BE PASSED FROM ACTIVITY 1 TO ACTIVITY 2 THROUGH ENTRY PARAMETERS.

```
task Activity_1_Task is                     task Activity_2_Task is
   ...                                         ...
   entry Get_Data (Data : out Float);       end Activity_2_Task;
end Activity_1_Task;

task body Activity_1_Task is                task body Activity_2_Task is
   ...                                         ...
begin                                       begin
   ...                                         ...
   loop                                        loop
      Activity_1 (Output_Data);                   Activity_1_Task.Get_Data (Input_Data);
      accept Get_Data (Data : out Float) do       Activity_2 (Input_Data);
         Data := Output_Data;                  end loop;
      end Get_Data;                         end Activity_2_Task;
      ...
      delay ...;
   end loop;
end Activity_1_Task;
```

- THE ENTRY ACTS AS A "GATE" FOR Activity_2_Task. GIVEN REASONABLE PROCESSOR LOAD, Activity_2_Task WILL USUALLY BE WAITING AT THE ENTRY CALL BY THE TIME Activity_1_Task EXECUTES ITS DELAY AND GOES ON TO REACH THE accept STATEMENT.

INSTRUCTOR NOTES

- BULLET 2:

  THE CHOICE MAY BE AN IMPLICIT CHOICE TO CONTINUE WITH THE CURRENTLY EXECUTING TASK EVEN THOUGH A NEW TASK HAS BECOME UNBLOCKED.

- BULLET 3:

  - ITEM 1: SLIDES 3-24, AND 3-25 EXPLAINED HOW TO USE DELAY STATEMENTS AND RENDEZVOUS TO ACHIEVE THIS EFFECT.

  - ITEM 2: THE NEXT SLIDE ADDRESSES THE PROBLEM OF ENSURING THAT THERE IS INDEED AMPLE CPU TIME AVAILABLE FOR ALL TASKS.

SCHEDULING OF SINGLE-THREAD TASKS

- BASED ON DELAY STATEMENTS, ENTRY CALLS, SIMPLE ACCEPT STATEMENTS, AND SELECTIVE WAITS, CERTAIN TASKS ARE <u>BLOCKED</u> (INELIGIBLE FOR EXECUTION) AT ANY TIME.
- ON THE FLY, THE RUNTIME SYSTEM CHOOSES AN UNBLOCKED TASK FOR EXECUTION.
  - THIS CHOICE REPLACES THE SCHEDULING DONE AT DESIGN TIME FOR CYCLIC EXECUTIVES.
  - IT MAY ENTAIL SOME RUNTIME OVERHEAD.
- CERTAIN CONSTRAINTS ENSURE THAT THE CHOICE RESULTS IN A DESIRABLE SCHEDULING OF TASKS:
  - PROPERLY DESIGNED DELAY STATEMENTS, ENTRY CALLS, AND ACCEPT STATEMENTS CAUSE TASKS TO BECOME UNBLOCKED AT THE REQUIRED FREQUENCY, BUT ONLY WHEN TASKS THAT MUST GO BEFORE IT HAVE BEEN EXECUTED.
  - IF THERE IS AMPLE CPU TIME FOR ALL THE REQUIRED PROCESSING, MOST TASKS WILL BE BLOCKED MOST OF THE TIME, SO EACH UNBLOCKED TASK WILL NORMALLY GET A CHANCE TO EXECUTE SHORTLY AFTER IT BECOMES UNBLOCKED.
  - TASKS CAN BE GIVEN <u>PRIORITIES</u>.
    - AN UNBLOCKED TASK OF HIGHER PRIORITY IS ALWAYS SELECTED FOR EXECUTION BEFORE ONE OF LOWER PRIORITY.
    - HIGHER-FREQUENCY LOOPS CAN BE GIVEN HIGHER PRIORITY.
    - THIS WILL TEND TO REDUCE JITTER. LOWER-FREQUENCY TASKS HAVE A GREATER OPPORTUNITY TO BE SCHEDULED WITHOUT FALLING BEHIND.

INSTRUCTOR NOTES

- BULLET 2:

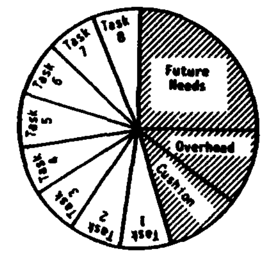
  - ITEM 4: $\underline{p}$ IS THE <u>AVERAGE</u> PERIOD BETWEEN ITERATIONS, SUBJECT TO JITTER.

  - ITEM 6: OPTIMIZATION TECHNIQUES CAN BE USED TO REDUCE THE VALUE OF $\underline{c}$ (i.e., TO ACCOMPLISH THE SAME PROCESSING WITH FEWER INSTRUCTIONS)

BUDGETING TIME

- PURE CYCLIC EXECUTIVE:
  - TOTAL RUNNING TIME OF ALL ACTIVITIES SCHEDULED FOR A GIVEN MINOR CYCLE MUST BE LESS THAN THE MINOR CYCLE LENGTH.
  - SIMPLE CRITERION, BUT ACHIEVING IT IS DIFFICULT.

- SINGLE-THREAD TASKS:
  - ALLOCATE A PORTION OF THE PROCESSOR TIME TO EACH TASK.
  - LEAVE A PORTION UNALLOCATED FOR:
    - RUNTIME SYSTEM OVERHEAD (AMOUNT BASED ON EXPERIENCE)
    - FUTURE ENHANCEMENTS
    - CUSHION FOR TEMPORARY PROCESSING OVERLOAD
  - LET c BE THE ESTIMATED MAXIMUM CPU TIME FOR ONE ITERATION OF THE TASK'S MAIN LOOP.
    - DOES NOT INCLUDE TIME SPENT IN DELAYS
    - DOES NOT INCLUDE TIME WAITING FOR RENDEZVOUS
  - LET p BE THE NOMINAL PERIOD OF THE LOOP (TIME BETWEEN ITERATIONS)
  - THE TASK MUST BE RUNNING c/p OF THE TIME TO DO THE PROCESSING REQUIRED FOR ONE ITERATION.
  - c/p MUST NOT BE GREATER THAN THE TASK'S ALLOCATED PORTION OF THE PROCESSOR TIME.
  - MORE COMPLICATED CRITERION, BUT RUNTIME SYSTEM IS RESPONSIBLE FOR ACHIEVING IT.




```
loop
   [computation]
   Next := Next + p;
   delay Next-Clock;
end loop;
```

c SECONDS CPU TIME
p SECONDS REAL TIME

AD-A145 093 ADA (TRADEMARK) TRAINING CURRICULUM REAL-TIME CONCEPTS 3/4
L303 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84
DAAB07-83-C-K514

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

INSTRUCTOR NOTES

THE CYCLIC EXECUTIVE APPROACH IS REALLY ONLY ADVANTAGEOUS FOR VERY SIMPLE, COMPLETELY SYNCHRONOUS PROBLEMS.

BOTH APPROACHES TO CYCLIC PROCESSING ARE AVAILABLE WITHIN Ada.

PURE CYCLIC EXECUTIVE VERSUS SINGLE-THREAD TASKS

ADVANTAGES OF PURE CYCLIC EXECUTIVE

- FAMILIARITY
- SIMPLICITY OF RUNTIME SYSTEM
- EFFICIENCY (LOW OVERHEAD)
- PREDICTABILITY
  - ACTIONS ALWAYS PERFORMED IN THE SAME ORDER
  - BEHAVIOR IS REPRODUCIBLE

ADVANTAGES OF SINGLE-THREAD TASKS

- MORE NATURAL PARTITION OF PROBLEM
  - ONE TASK PER REAL-WORLD ACTIVITY
  - WELL-DEFINED INTERFACES AND DATA FLOW
- FLEXIBILITY
  - SCHEDULING OF THE PROCESSOR CAN CHANGE DYNAMICALLY TO MEET VARYING LOADS
    - URGENT TASKS CAN "OVERFLOW" WHEN NECESSARY, ALLOWING LESS URGENT TASKS TO CATCH UP LATER
    - NO PREIMPOSED IDLE PERIODS WHEN THERE IS WORK TO BE DONE
  - CYCLE LENGTHS CAN BE PRECISELY FITTED TO NATURAL PROBLEM REQUIREMENTS
    - NEED NOT BE MINOR CYCLE LENGTH TIMES A POWER OF TWO
    - BY NOT CYCLING MORE FREQUENTLY THAN NECESSARY, WE CONSERVE PROCESSING POWER
  - EVENT-DRIVEN ASYNCHRONOUS PROCESSING EASILY ACCOMMODATED
- EASE OF DESIGN
  - NO CYCLE ALLOCATION PROBLEM
  - PROCESSOR SCHEDULING HANDLED BY RUNTIME SYSTEM

INSTRUCTOR NOTES

- ALLOW 30 MINUTES FOR THIS SECTION

- THE MAIN MESSAGES ARE

  - TUNING IS OFTEN NECESSARY TO MEET REAL-TIME CONSTRAINTS.

  - TYPICALLY, PROGRAMS SPEND MOST OF THEIR TIME EXECUTING A SMALL PART OF THE PROGRAM.

  - WHICH PARTS OF A PROGRAM SHOULD BE TUNED OR WHETHER TUNING IS EVEN NECESSARY GENERALLY CAN ONLY BE DETERMINED BY RUNNING THE PROGRAM.

  - PREMATURE TUNING MAKES PROGRAMS HARDER TO DEVELOP, UNDERSTAND AND MODIFY, AND OFTEN DOES NOT CONTRIBUTE TO IMPROVED PERFORMANCE.

SECTION 4

IMPROVING PERFORMANCE THROUGH TUNING

INSTRUCTOR NOTES

- BULLET 1 — AN ONBOARD COMPUTER SYSTEM FOR AN AIRCRAFT MUST REPORT AN INCOMING MISSILE IN A TIMELY MANNER. THE ONBOARD SYSTEM MIGHT ALSO BE SUBJECT TO WEIGHT CONSTRAINT WHICH PREVENTS ADDITIONAL MEMORY FROM BEING ADDED (BECAUSE OF THE HOUSING).

- BULLET 2

  - ITEM 1 - EMPHASIZE WORKING WE WILL SEE IN A FEW SLIDES WHY WE WANT TO START WITH A WORKING SYSTEM.

  - ITEM 2 - THESE STEPS NEED TO BE PERFORMED SEVERAL TIMES.
    - TO ENSURE THAT CHANGES DID IMPROVE THE SYSTEM, AND
    - TO FIND OUT IF ADDITIONAL PARTS OF THE PROGRAM NEED ATTENTION

  - ITEM 3 - WHILE PROGRAM CLARITY IS IMPORTANT, A WEATHER "PREDICTION" PROGRAM THAT MERELY CONFIRMS YESTERDAY'S WEATHER IS UNREASONABLE.

REAL-TIME REQUIREMENTS

- EMBEDDED REAL-TIME SYSTEMS OFTEN MUST SATISFY CONSTRAINTS
  - EXECUTION TIME
  - STORAGE SIZE

- TO MEET THESE CONSTRAINTS, SUCH SYSTEMS MAY NEED TO BE TUNED.
  - SYSTEM <u>TUNING</u> IS THE SEQUENCE OF ACTIONS APPLIED TO A <u>WORKING</u> SYSTEM TO PRODUCE A MORE "EFFICIENT" <u>WORKING</u> SYSTEM.
  - TUNING INCLUDES REPEATEDLY
    1. DETERMINING PARTS OF THE PROGRAM THAT AFFECT <u>OVERALL</u> SYSTEM PERFORMANCE.
    2. MODIFYING ONE MORE OF
       - SEQUENCE OF STATEMENTS EXECUTED
       - DATA STRUCTURES USED
       - ALGORITHMS USED
  - TUNING MAY SACRIFICE PROGRAM CLARITY

- THIS SECTION SUGGESTS
  - WAYS TO DETERMINE WHEN PERFORMANCE PROBLEMS EXIST
  - WHEN TO CONSIDER PROGRAM EFFICIENCY

INSTRUCTOR NOTES

- BULLET 1
  - WE GIVE AN EXAMPLE OF PREMATURE OPTIMIZATION PROBLEMS SHORTLY
  - THE QUOTE IS FROM KNUTH'S ARTICLE STRUCTURED PROGRAMMING WITH GO TO STATEMENTS. THE STUDY IS DESCRIBED IN AN EMPIRICAL STUDY OF FORTRAN PROGRAMS. (SEE BIBLIOGRAPHY).
- BULLET 2
  - ITEM 1 IS OBTAINED FROM KNUTH'S FORTRAN PROGRAM STUDY.
  - ITEM 2 IMPROVING THE PERFORMANCE OF THE 3% AFFECT MORE THAN 50% OF THE EXECUTION TIME.

HOW MUCH OF A PROGRAM SHOULD BE TUNED

- DONALD KNUTH PERFORMED STUDIES ON FORTRAN PROGRAMS. HE CONCLUDES

  "... MOST OF THE RUNNING TIME IN NON-IO-BOUND PROGRAMS IS CONCENTRATED IN ABOUT 3% OF THE SOURCE TEXT ... WE SHOULD FORGET ABOUT SMALL EFFICIENCIES, SAY ABOUT 97% OF THE TIME: PREMATURE OPTIMIZATION IS THE ROOT OF ALL EVIL."

- CONCENTRATE TUNING ON THE 3%.
  - ACCOUNTS FOR OVER 50% OF PROGRAM EXECUTION.
  - TUNING HERE PROVIDES THE BEST RETURN.
    - SPEEDING UP A CRITICAL LOOP BY 10% MAY SPEED UP ENTIRE PROGRAM BY NEARLY 10%
    - SPEEDING UP OTHER CODE BY 50% MAY HAVE NO MEASURABLE EFFECT

- NEED TO FIND THE PORTIONS OF THE PROGRAM THAT MAKE UP THE 3%.

- PROFILING CAN AFFECT OVERALL EXECUTION TIME SINCE CODE IS INSERTED IN THE PROGRAM. THIS IS ESPECIALLY TRUE FOR FREQUENCY COUNTING. THE STATISTICAL PROFILE IS LESS ACCURATE, BUT IT CAN BE MORE REALISTIC IN THAT IT SHOWS HOW MUCH TIME THE PROGRAM SPENT IN SYSTEM ROUTINES. IF FIXED INTERVALS ARE USED FOR THE STATISTICAL PROFILE, THEN SEVERAL RUNS WITH DIFFERENT INTERVALS SHOULD BE OBTAINED. THIS AVOIDS THE PROBLEM OF THE SAMPLING GETTING IN SYNC WITH A LOOP.

- A PROFILE PROGRAM CREATES A COPY OF THE SOURCE AND ADDS STATEMENTS TO DO THE PROFILE. A COMPILER ADDS OBJECT CODE TO DO THE PROFILE.

- BULLET 3 - ITEM 1

  - SUBITEM 1 - THIS KIND OF ERROR MIGHT BE THE RESULT OF AN INCORRECTLY FORMULATED CONDITIONAL EXPRESSION IN AN IF-STATEMENT, WHILE-STATEMENT, ETC.

  - SUBITEM 2 - DURING CODING, THE PROGRAMMER MAY HAVE FAILED TO REALIZE THAT A PARTICULAR CASE CANNOT HAPPEN. FOR EXAMPLE, TESTING IF A VARIABLE IS NEGATIVE AFTER IT HAS BEEN SQUARED.

  - SUBITEM 3 - THIS MIGHT REQUIRE ADDITIONAL RUNS (POSSIBLE WITH ADDITIONAL TEST DATA). THIS DOESN'T NEED TO BE A POSITIVE PATH. IT COULD BE AN ERROR PATH. ERROR PATHS SHOULD BE CONSIDERED WHEN TUNING. A NUCLEAR POWER PLANT CONTROL PROGRAM SHOULD NOT "SLOW DOWN" IF AN ERROR OCCURS.

  - ITEM 2 - SEE THE TABLE LOOK UP SLIDE #9.

- BULLET 4 - THE NEXT SLIDE DESCRIBES WHY

DETERMINE WHERE TO OPTIMIZE

- THE PARTS TO TUNE CAN BE DETERMINED USING PROFILES.
  - FREQUENCY PROFILE
    - COUNTERS ARE INSERTED IN THE PROGRAM
    - SHOWS HOW OFTEN EACH STATEMENT IS EXECUTED
    - SHOWS HOW MUCH TIME SPENT IN EACH SUBPROGRAM
  - STATISTICAL PROFILE
    - PROGRAM IS FREQUENTLY INTERRUPTED WITH LOCATION IN PROGRAM NOTED
    - SHOWS HOW OFTEN THE PROGRAM WAS IN A CERTAIN RANGE OF STATEMENTS (INSTRUCTIONS)
- PROFILES MAY BE PROVIDED BY
  - COMPILER
  - PROFILE PROGRAM
  - THE ALS Ada COMPILER PROVIDES FOR BOTH KINDS OF PROFILES
- SIDE BENEFITS
  - SHOWS PARTS OF PROGRAM THAT ARE NEVER EXECUTED, CALLING ATTENTION TO:
    - INCORRECTLY FORMULATED CONDITIONS
    - UNREACHABLE CODE
    - INCOMPLETE TESTING (E.G. ERROR PATHS)
  - SHOWS POSSIBLE RELATIONSHIPS BETWEEN PARTS OF PROGRAM.
- INITIAL PROFILES SHOULD BE OBTAINED BEFORE TUNING STARTS.

INSTRUCTOR NOTES

- THIS IS THE FIRST OF TWO SLIDES GIVING AN EXAMPLE OF HOW PREMATURE TUNING CAN RESULT IN WASTED EFFORT

A PENALTY OF PREMATURE TUNING

- IN HIS BOOK, <u>WRITING EFFICIENT PROGRAMS</u> (SEE BIBLIOGRAPHY) BENTLEY RELATES THE FOLLOWING STORY:

- A FORTRAN COMPILER WAS BEING ENHANCED IN EARLY 1960's SUBJECT TO THE CONSTRAINT THAT USERS WOULD NOT NOTICE AN INCREASE IN COMPILATION TIME.

- A PARTICULAR SUBPROGRAM WITHIN THE COMPILER WAS RARELY USED. THE PROGRAMMER DOING THE ENHANCEMENTS ESTIMATED THAT

  - 1% OF THE COMPILATIONS USED THE ROUTINE
  - ROUTINE USED AT MOST ONCE PER COMPILATION

- SINCE THE SUBPROGRAM WAS VERY SLOW, IT WAS TUNED.

  - TUNING TOOK 1 WEEK
  - SQUEEZED "...EVERY LAST UNNEEDED CYCLE OUT OF THE SUBPROGRAM."

- MODIFIED COMPILER RAN FAST ENOUGH.

INSTRUCTOR NOTES

- THIS IS A FAIRLY COMMON SITUATION. PROGRAMMERS ALWAYS "THINK" THEY KNOW WHERE TUNING SHOULD OCCUR. PROFILES TELL US WHERE TUNING SHOULD OCCUR.

- WE WILL RETURN TO PROBLEMS OF PREMATURE TUNING.

MUCH ADO ABOUT NOTHING

- AFTER 2 YEARS EXTENSIVE USE
  - COMPILER REPORTED AN INTERNAL ERROR DURING COMPILATION.
  - THE ERROR WAS IN THE PROLOGUE (STARTING CODE) OF THE "CRITICAL" SUBPROGRAM.
  - ERROR HAD EXISTED DURING ENTIRE LIFE OF THE SUBPROGRAM.

- CONCLUSION REACHED WAS
  - SUBPROGRAM NEVER CALLED IN THE MORE THAN 100,000 COMPILATIONS EXECUTED IN THE LIFE OF THE "ENHANCED" COMPILER.
  - THE WEEK'S EFFORT AT TUNING THE SUBPROGRAM WAS <u>MUCH ADO ABOUT NOTHING</u>.

- IN GENERAL, PROGRAMMERS HAVE A <u>GOOD</u> TRACK RECORD AT BEING <u>VERY BAD</u> AT GUESSING WHICH PARTS OF A PROGRAM NEED TO BE TUNED.
  - KNUTH HAS FOUND THAT

    "IT IS OFTEN A MISTAKE TO MAKE A PRIORI JUDGEMENTS ABOUT WHAT PARTS OF A PROGRAM ARE REALLY CRITICAL, SINCE THE UNIVERSAL EXPERIENCE OF PROGRAMMERS WHO HAVE BEEN USING MEASUREMENT TOOLS HAS BEEN THAT THEIR INTUITIVE GUESSES FAIL."

- PROFILES SHOULD BE USED TO FIND WHERE PROGRAM TUNING IS NEEDED.

INSTRUCTOR NOTES

- THIS SLIDE STATES THE PROBLEMS WITH PREMATURE OPTIMIZATION IN GENERAL.

- BULLETS 3, ITEM 2 - AN EXAMPLE IS EXPLOITING THE BIT REPRESENTATION OF THE WORD PROVIDED BY A PARTICULAR SENSOR. ALGORITHM MUST CHANGE WHEN A NEW SENSOR IS INSTALLED.

PROBLEMS WITH PREMATURE TUNING

- MISDIRECTED EFFORT
  - THE FORTRAN COMPILER EXAMPLE SHOWS THAT EFFORT MIGHT BE EXPENDED NEEDLESSLY.
- LESS RELIABLE
  - MANY TIMES TUNING INVOLVES COMPLEX OR SUBTLE LOGIC.
  - COMPLEX CODE IS MORE DIFFICULT TO WRITE AND MORE ERROR-PRONE.
- LESS MAINTAINABLE
  - PROGRAM MORE DIFFICULT TO UNDERSTAND.
  - PROGRAM TAKES ADVANTAGE OF PROPERTIES OF THE PROBLEM INCREASING THE SENSITIVITY OF THE CODE TO CHANGES IN THE PROBLEM.
  - GREATER CHANCE OF INTRODUCING ERRORS DUE TO COMPLEX/HIDDEN INTERACTIONS, OR DEPENDENCIES ON COMPLICATED ASSUMPTIONS.
- INCREASED INTERACTION CAN INCREASE RECOMPILATION DEPENDENCIES.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS THE DRAMATIC RESULTS THAT CAN BE OBTAINED BY USING A PROFILE. THIS IS NOT ATYPICAL. OF COURSE ADDITIONAL TUNING WILL NOT BE AS "EFFECTIVE" BUT IT WILL STILL IMPROVE PERFORMANCE.

- BULLETS 4 AND 5 - EMPHASIZE 20 LINES OF CODE WITH A FEW HOURS WORK.

- BULLET 6 - TELL THE CLASS THAT OBTAINING ANOTHER PROFILE NOT ONLY VERIFIES THAT TUNING CHANGES HAVE THE DESIRED EFFECT, BUT IT CAN ALSO POINT OUT ADDITIONAL PARTS OF THE PROGRAM THAT CAN BE TUNED.

PROFILE OF A SUCCESS STORY

- BENTLEY RELATES THE FOLLOWING STORY
- C. VAN WYK (OF BELL LABS) PROFILED AN INTERPRETER HE HAD DEVELOPED.
    - RAN 10 TEST CASES THAT EXECUTED EVERY PROGRAM PATH.
- THE PROFILE SHOWED
    - 70% OF THE EXECUTION TIME WAS SPENT IN THE SYSTEM MEMORY ALLOCATION SUBPROGRAM.
    - LOOKING AT THIS ROUTINE, VAN WYK FOUND
        - MOST OF THE TIME SPENT ALLOCATING ONE PARTICULAR KIND OF RECORD -- 68,000 TIMES
        - NEXT MOST POPULAR RECORD ALLOCATED 2000 TIMES
- VAN WYK ADDED ABOUT 20 LINES OF CODE TO KEEP FREE RECORDS OF THIS TYPE ON A QUEUE THAT THE PROGRAM MANAGED ITSELF.
    - THE MODIFIED PROGRAM'S RUNTIME DECREASED TO 45%.
    - MEMORY ALLOCATION DECREASED TO 30% OF PROGRAM EXECUTION.
- ALL OF THIS WAS ACCOMPLISHED IN A FEW HOURS ONE AFTERNOON.
- SIDE BENEFIT-OBTAINING PROFILE OF IMPROVED VERSION
    - POINTED OUT PLACES FOR ADDITIONAL TUNING.
    - THESE WERE DETECTED BECAUSE PROFILE WAS NO LONGER OVERWHELMED BY MEMORY ALLOCATION.

INSTRUCTOR NOTES

- TUNING CAN MEAN USING A BETTER ALGORITHM (BINARY SEARCH RATHER THAN LINEAR SEARCH) OR IT CAN MEAN USING EFFICIENT CODING TECHNIQUES. IN EITHER CASE, WE NEED TO BE ABLE TO UNDERSTAND THE PROGRAM BEFORE WE CAN TUNE IT. WE ALSO NEED TO UNDERSTAND THE DESIGN BEFORE WE CAN PREPARE AN UNDERSTANDABLE AND CORRECT IMPLEMENTATION.

- MUST START WITH A CORRECT IMPLEMENTATION. IT MAKES NO SENSE TO EXPEND EFFORT TO MAKE A SYSTEM FAIL FASTER.

- ANYTIME YOU START TINKERING WITH A PROGRAM YOU OPEN UP THE POSSIBILITIES OF INTRODUCING ERRORS.

STEPS TO TAKE BEFORE TUNING

- MUST START WITH A CORRECT AND EASILY UNDERSTOOD DESIGN.

- PRODUCE A SYSTEM THAT IS A CORRECT AND EASILY UNDERSTOOD IMPLEMENTATION OF THE DESIGN.

  - IN ORDER TO TUNE A SYSTEM, IT MUST BE UNDERSTOOD.
  - TUNING OFTEN MAKES A SYSTEM MORE DIFFICULT TO UNDERSTAND, HENCE MAINTAIN.
  - TUNING A SYSTEM OPENS UP THE POSSIBILITY OF INTRODUCING ERRORS.

- DETERMINE IF PERFORMANCE IS AN ISSUE.

  "THERE IS NO SENSE IN MAKING A PROGRAM RUN FASTER IF THE RESULT IS MERELY TO INCREASE THE PROPORTION OF UNUSED CPU CYCLES"
  -- M.A. Jackson, PRINCIPLES OF PROGRAM DESIGN.

- USE PROFILES TO PINPOINT BOTTLENECKS.

  - LEAVE THE REST OF THE SYSTEM ALONE.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS WHY YOU CANNOT ACCEPT PROFILE RESULTS BLINDLY

USE COMMON SENSE IN USING PROFILES

- WHEN YOU INTERPRET RESULTS FROM PROFILES
  - LOOK AT THE CONTEXT IN WHICH THE TIME CONSUMING PORTIONS APPEAR.
  - IF A GREAT DEAL OF TIME IS SPENT IN SMALL SUBPROGRAMS, THE PROBLEM MIGHT BE THAT THE SUBPROGRAM IS CALLED SO OFTEN.

- SUPPOSE A TABLE LOOK UP PACKAGE PROFILE SHOWS THAT 90% OF THE TIME IS SPENT IN A VARIABLE LENGTH STRING COMPARE SUBPROGRAM.

- TWO APPROACHES CAN BE USED TO ANALYZE THE RESULTS.
  - NAIVE APPROACH
    - SPEED UP THE COMPARE SUBPROGRAM
    - MAYBE GET FACTOR OF TWO SPEEDUP
  - MORE SOPHISTICATED APPROACH
    - STUDY HOW THE COMPARE SUBPROGRAM IS USED
    - MIGHT FIND TABLE LOOK UP PACKAGE USING A LINEAR SEARCH
    - REPLACING THE LINEAR SEARCHING WITH HASHING MIGHT GET A FACTOR TEN SPEEDUP.

INSTRUCTOR NOTES

- EFFICIENCY IS IMPORTANT, BUT IT'S JUST ONE OF THE PROGRAMMING CONSIDERATIONS THAT RESULT IN GOOD PROGRAMMING. STRUCTURE IS IMPORTANT, BUT AGAIN IT'S JUST ONE OF THE CONSIDERATIONS. THE SUCCESSFUL PROGRAMMER MUST MAINTAIN THE PROPER PERSPECTIVE ON BOTH CONSIDERATIONS.

- THIS IS A GOOD PLACE TO URGE THE CLASS TO READ BENTLEY'S BOOK AND TO SUGGEST THAT HIS <u>PROGRAMMING PEARLS</u> COLUMN SHOULD BE CONSIDERED MUST READING.

KEEPING THE PROPER PERSPECTIVE

IN PROGRAMMING PEARLS, BENTLEY WRITES

> "SOME PROGRAMMERS PAY TOO MUCH ATTENTION TO EFFICIENCY; BY WORRYING TOO SOON ABOUT LITTLE 'OPTIMIZATIONS' THEY CREATE RUTHLESSLY CLEVER PROGRAMS THAT ARE INSIDIOUSLY DIFFICULT TO MAINTAIN. OTHERS PAY TOO LITTLE ATTENTION; THEY END UP WITH BEAUTIFULLY STRUCTURED PROGRAMS THAT ARE UTTERLY INEFFICIENT AND THEREFORE USELESS. ONE HAS TO KEEP PERSPECTIVE ON EFFICIENCY: IT IS JUST ONE OF MANY PROBLEMS, BUT IT IS SOMETIMES VERY IMPORTANT."

INSTRUCTOR NOTES

- TAKE 10 MINUTES FOR THIS SECTION. IT SUMMARIZES THE FOUR SECTIONS OF THE COURSE.

- THE TITLE OF EACH SLIDE IS THE NAME OF THE SECTION BEING SUMMARIZED.

SECTION 5

SUMMARY

VG 831

INSTRUCTOR NOTES

- BULLET 3 - EMPHASIZE THAT THE RUNTIME SYSTEM CAN BE MODIFIED FOR THE PROBLEM BEING SOLVED.

CONCURRENT PROGRAMMING CONCEPTS

- CONCURRENCY PROVIDES A MORE NATURAL WAY OF LOOKING AT CERTAIN PROBLEMS.
  - MANAGING SIMULTANEOUS REAL-TIME ACTIVITIES
  - SIMULATING SIMULTANEOUS ACTIVITIES
  - LOGICALLY DECOMPOSING A PROBLEM INTO STREAM TRANSFORMATIONS
- CONCURRENCY INTRODUCES POTENTIAL PROBLEMS.
  - SIMULTANEOUS UPDATE
  - DEADLOCK
  - STARVATION
  - TASK COOPERATION
- Ada USES A RUNTIME SYSTEM TO IMPLEMENT ITS CONCURRENCY FEATURES.
  - RUNTIME SYSTEM IS HIDDEN FROM THE PROGRAMMER.
  - RUNTIME SYSTEM CAN BE TAILORED TO HELP MEET SCHEDULING AND/OR PERFORMANCE CONSTRAINTS.

INSTRUCTOR NOTES

- BULLET 3 - EMPHASIZE THIS.

Ada TASKING FEATURES

- TASK OBJECTS ARE DATA OBJECTS IN A TASK TYPE.
- ALL FEATURES ARE BASED ON THE RENDEZVOUS CONCEPT.
  - COMMUNICATION BETWEEN TWO TASKS TAKES PLACE ONLY WHEN BOTH ARE READY.
  - TWO TASKS COMMUNICATE WHEN AN accept STATEMENT ACCEPTS AN ENTRY CALL.
  - IF ONE TASK IS READY TO COMMUNICATE AND THE OTHER IS NOT, THEN THE READY TASK WAITS FOR THE OTHER TASK.
- TASKS CAN EXERCISE VARYING DEGREES OF CONTROL OVER ENTRY CALLS THEY MAKE AND ACCEPT.
  - SELECTIVE WAIT STATEMENTS
    - GUARDS
    - DELAY ALTERNATIVE
    - ELSE PARTS
  - TIMED ENTRY CALLS
  - CONDITIONAL ENTRY CALLS
- HARDWARE INTERRUPTS CAN BE MADE TO LOOK LIKE AN ENTRY CALL.
  - PROVIDES ABSTRACT VIEW OF HARDWARE INTERRUPTS.
  - ALLOWS DEVICE DRIVERS TO BE WRITTEN AT HIGH LEVEL OF ABSTRACTION.
- PRIORITIES CAN BE ASSIGNED TO TASKS.
  - CAN BE USED TO
    - INDICATE RELATIVE DEGREES OF URGENCY
    - RESOLVE CONTENTION FOR CPU USE
  - CANNOT BE USED TO SYNCHRONIZE TASKS

INSTRUCTOR NOTES

- EMPHASIZE BULLETS 4 AND 5.

FUNDAMENTAL TASK DESIGNS

- SERVER AND USER TASKS
  - ONE TASK (THE USER) CALLS ANOTHER TO OBTAIN SOME SERVICE FROM THAT TASK.
  - A TASK (THE SERVER) ACCEPTS AN ENTRY CALL IN ORDER TO PROVIDE SOME SERVICE TO THE CALLING TASK.
  - THE CALLING TASK "KNOWS" WHOM ITS CALLING.
  - THE CALLED TASK DOES NOT "KNOW" WHO CALLED IT.

- MONITORS
  - CAN BE USED TO SOLVE THE SIMULTANEOUS UPDATE PROBLEM.
  - PROVIDES A LIMITED SET OF OPERATIONS THROUGH WHICH SEVERAL CONCURRENT TASKS CAN MANIPULATE DATA HIDDEN INSIDE THE MONITOR.

- MESSAGE BUFFERS
  - ALLOW TASKS TO COMMUNICATE WITHOUT WAITING FOR EACH OTHER.

- RENDEZVOUS CAN BE USED AS PRIMITIVE TO BUILD MORE ELABORATE MECHANISMS FOR TASK COMMUNICATION.
  - FREQUENTLY USED MECHANISMS CAN ALSO BE PROVIDED DIRECTLY BY A RUNTIME SYSTEM FOR THE SAKE OF EFFICIENCY.

- STREAM ORIENTED DESIGN
  - SEQUENTIAL PROBLEM CAN SOMETIMES BE SIMPLIFIED BY DECOMPOSING INTO TASKS VIEWED AS PROCEEDING IN PARALLEL.
  - EACH TASK MODELS A STREAM TRANSFORMATION
  - TASKS USED TO SOLVE PROBLEMS NOT INHERENTLY CONCURRENT

- CYCLIC EXECUTIVES
  - TRADITIONAL APPROACH CAN BE CODED DIRECTLY IN Ada.
  - Ada PROVIDES MORE NATURAL WAY
    - ONE TASK PER THREAD OF CONTROL
    - EASIER TO UNDERSTAND
    - LESS ERROR-PRONE

INSTRUCTOR NOTES

IMPROVING PERFORMANCE THROUGH TUNING

- A PROGRAM MAY BE MADE EFFICIENT THROUGH TUNING
- STUDIES SHOW MOST TIME SPENT <u>IN SMALL</u> PORTION OF PROGRAM
  - TUNE THESE "HOT SPOTS" ONLY
  - USE PROFILES TO DETERMINE "HOT SPOTS"
- START WITH UNDERSTANDABLE CORRECT WORKING PROGRAM
  - MUST UNDERSTAND PROGRAM
  - TUNING INTRODUCES COMPLEXITY
- PREMATURE TUNING
  - RESULTS IN WASTED EFFORT
  - ERROR-PRONE PROGRAMS
  - INCREASED COMPILATION DEPENDENCIES

INSTRUCTOR NOTES

"THIS CONCLUDES THE COURSE"

REAL-TIME PROGRAMMING IN Ada

- Ada IS A VIABLE LANGUAGE FOR REAL-TIME PROGRAMMING
  - STANDARD PROBLEMS SUCH AS CYCLIC EXECUTIVES CAN BE SOLVED IN Ada.
  - THE RUNTIME SYSTEM CAN BE ADAPTED TO SUPPORT A PARTICULAR ENVIRONMENT.

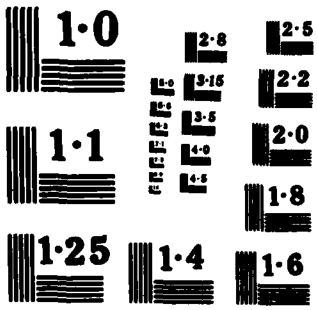
- IF A REAL-TIME PROBLEM CAN BE SOLVED EFFECTIVELY, IT CAN BE SOLVED EFFECTIVELY IN Ada.

AD-A145 093 ADA (TRADEMARK) TRAINING CURRICULUM REAL TIME CONCEPTS L303 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07 83 C K514 4/4

UNCLASSIFIED F/G 9/2 NL

INFORMATION

END
DATE
FILMED
11 84


1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

SUPPLEMENTARY

INFORMATION

DEPARTMENT OF THE ARMY
HEADQUARTERS US ARMY COMMUNICATIONS-ELECTRONICS COMMAND
AND FORT MONMOUTH
FORT MONMOUTH, NEW JERSEY 07703

REPLY TO
ATTENTION OF:

15 OCT 1984

Center for Tactical Computer Systems

Ms. Madeline Crumbacker
Defense Tactical Information Center
Cameron Station
Alexandria, Virginia 22314

Dear Ms. Crumbacker:

As per phone conversation with Ms. Andrea Cappellini, CENTACS on 11 October 1984, a copyright statement has been omitted on documents sent to DTIC and NTIS. Enclosed please find the copyright statement (Encl 1) that must appear in the enclosed list of document (Encl 2). If you have any questions, please contact Ms. Cappellini at 201-544-4280.

Sincerely,

Dennis J. Turner
for JAMES E. SCHELL
Director, CENTACS

DATE
ILMED
-8